Oracle® Fusion Middleware

Programming Advanced Features of JAX-WS Web Services for Oracle WebLogic Server

12*c* Release 1 (12.1.1)

E24965-02

January 2012

Documentation for software developers that describes how to use Web Services Reliable Messaging (WS-ReliableMessaging) for JAX-WS Web services to enable a client application to reliably invoke a Web service running on another application server, with guaranteed message delivery between the two endpoints.

ORACLE

Oracle Fusion Middleware Programming Advanced Features of JAX-WS Web Services for Oracle WebLogic Server, 12*c* Release 1 (12.1.1)

E24965-02

Contents

5 Roadmap for Developing Reliable Web Services and Clients

6 Using Web Services Reliable Messaging

7 Managing Web Service Persistence

16 Handling Exceptions Using SOAP Faults

17 Creating and Using SOAP Message Handlers

Preface

This preface describes the document accessibility features and conventions used in this guide—*Programming Advanced Features of JAX-WS Web Services for Oracle WebLogic Server*

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introduction

This chapter provides a summary table advanced features for software developers who program WebLogic Web services using Java API for XML Web Services (JAX-WS).

Table 1–1 Programming Advanced Features Using JAX-WS

Advanced Features	Description
Chapter 2, "Using Web Services Addressing"	Describes how to use Web Services Addressing (WS-Addressing) to address Web services and their associated messages in a transport-neutral way.
Chapter 3, "Roadmaps for Developing Web Service Clients"	Review best practices for developing Web service clients.
Chapter 4, "Invoking Web Services Asynchronously"	Invoke a Web service asynchronously.
Chapter 5, "Roadmap for Developing Reliable Web Services and Clients"	Review best practices for developing asynchronous and reliable applications together.
Chapter 6, "Using Web Services Reliable Messaging"	Use Web service reliable messaging to enable an application running on one application server to *reliably* invoke a Web service running on another application server, assuming that both servers implement the WS-ReliableMessaging specification.
Chapter 7, "Managing Web Service Persistence"	Manage persistence for Web services. Web service persistence is used by advanced features to support long running requests and to survive server restarts.
Chapter 8, "Configuring Message Buffering for Web Services"	Configure message buffering for Web services.
Chapter 9, "Managing Web Services in a Cluster"	Review best practices for using Web services in a cluster.
Chapter 10, "Using Web Services Atomic Transactions"	Use Web services atomic transactions to enable interoperability with other external transaction processing systems.
Chapter 11, "Publishing a Web Service Endpoint"	Publish a Web service endpoint at runtime, without deploying the Web service.
Chapter 12, "Using Callbacks"	Notify a client of a Web service that an event has happened by programming a callback.
Chapter 13, "Optimizing Binary Data Transmission"	Send binary data using MTOM/XOP and/or streaming SOAP attachments to optimize transmission of binary data.
Chapter 14, "Developing Dynamic Proxy Clients"	Invoke a Web service based on a service endpoint interface (SEI) dynamically at run-time without using `clientgen`.

Table 1–1 (Cont.) Programming Advanced Features Using JAX-WS

Advanced Features	Description
Chapter 15, "Using XML Catalogs"	Use XML catalogs to resolve network resources to versions that are stored locally.
Chapter 16, "Handling Exceptions Using SOAP Faults"	Handle SOAP faults and exceptions.
Chapter 17, "Creating and Using SOAP Message Handlers"	Create and configure SOAP message handlers for a Web service.
Chapter 19, "Operating at the XML Message Level"	Develop Web service provider-based endpoints and dispatch clients to operate at the XML message level.
Chapter 20, "Programming Web Services Using XML Over HTTP"	Program Web services using XML over HTTP.
Chapter 21, "Programming Stateful JAX-WS Web Services Using HTTP Session"	Create a Web service that maintains state between service calls.
Appendix A, "Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection"	Review the pre-packaged WS-Policy files that contain typical reliable messaging assertions that you can use to support reliable messaging.
Appendix B, "Example Client Wrapper Class for Batching Reliable Messages"	Provides an example client wrapper class that can be used for batching reliable messaging.

Note: The JAX-WS implementation in Oracle WebLogic Server is extended from the JAX-WS Reference Implementation (RI) developed by the Glassfish Community (see `https://jax-ws.dev.java.net/`). All features defined in the JAX-WS specification (JSR-224) are fully supported by Oracle WebLogic Server.

The JAX-WS RI also contains a variety of extensions, provided by Glassfish contributors. Unless specifically documented, JAX-WS RI extensions are not supported for use in Oracle WebLogic Server.

For an overview of WebLogic Web services, standards, samples, and related documentation, see *Introducing Web Services*.

JAX-WS supports Web Services Security (WS-Security) 1.1. For information about WebLogic Web service security, see *Securing WebLogic Web Services for Oracle WebLogic Server*.

2

Using Web Services Addressing

This chapter describes how to use Web Services Addressing (WS-Addressing) for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

2.1 Overview of WS-Addressing

WS-Addressing provides a transport-neutral mechanism to address Web services and their associated messages. Using WS-Addressing, endpoints are uniquely and unambiguously defined in the SOAP header.

WS-Addressing provides two key components that enable transport-neutral addressing, including:

- **Endpoint reference (EPR)**—Communicates the information required to address a Web service endpoint.
- **Message addressing properties**—Communicates end-to-end message characteristics, including addressing for source and destination endpoints and message identity, that allows uniform addressing of messages independent of the underlying transport.

  Message addressing properties can include one or more of the properties defined in Table 2–1. All properties are optional except `wsa:Action`.

Table 2–1 WS-Addressing Message Addressing Properties

Component	Description
`wsa:To`	Destination. If not specified, the destination defaults to `http://www.w3.org/2005/08/addressing/anonymous`.
`wsa:From`	Source endpoint.
`wsa:ReplyTo`	Reply endpoint. If not specified, the reply endpoint defaults to `http://www.w3.org/2005/08/addressing/anonymous`.
`wsa:FaultTo`	Fault endpoint.

Table 2–1 (Cont.) WS-Addressing Message Addressing Properties

Component	Description
`wsa:Action`	Required action. This property is required when WS-Addressing is enabled. It can be implicitly or explicitly configured, as described in Section 2.4, "Associating WS-Addressing Action Properties."
`wsa:MessageID`	Unique ID of the message.
`wsa:RelatesTo`	Message ID to which the message relates. This element can be repeated if there are multiple related messages. You can specify `RelationshipType` as an attribute of this property, which defaults to `http://www.w3.org/2005/08/addressing/reply`.
`wsa:ReferenceParameters`	Reference parameters that need to be communicated.

Example 2–1 shows a SOAP 1.2 request message sent over HTTP 1.2 with WS-Addressing enabled. As shown in **bold**, WS-Addressing provides a transport-neutral mechanism for defining a unique ID for the message (`wsa:MessageID`), the destination (`wsa:To`) endpoint, the reply endpoint (`wsa:ReplyTo`), and the required action (`wsa:Action`).

Example 2–1 SOAP 1.2 Message With WS-Addressing—Request Message

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
            xmlns:wsa="http://www.w3.org/2005/08/addressing/">
  <S:Header>
    <wsa:MessageID>
      http://example.com/someuniquestring
    </wsa:MessageID>
    <wsa:ReplyTo
      <wsa:Address>http://example.com/Myclient</wsa:Address>
    </wsa:ReplyTo>
    <wsa:To>
      http://example.com/fabrikam/Purchasing
    </wsa:To>
    <wsa:Action>
      http://example.com/fabrikam/SubmitPO
    </wsa:Action>
  <S:Header>
  <S:Body>
    ...
   </S:Body>
</S:Envelope>
```

A response to this message may appear as shown in Example 2–2. The `RelatesTo` property correlates the response message with the original request message.

Example 2–2 SOAP 1.2 Message Without WS-Addressing—Response Message

```
<S:Envelope
  xmlns:S="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <S:Header>
    <wsa:MessageID>http://example.com/someotheruniquestring</wsa:MessageID>
    <wsa:RelatesTo>http://example.com/someuniquestring</wsa:RelatesTo>
    <wsa:To>http://example.com/MyClient/wsa:To>
    <wsa:Action>
      http://example.com/fabrikam/SubmitPOAck
```

```
    </wsa:Action>
  </S:Header>
  <S:Body>
    ...
  </S:Body>
</S:Envelope>
```

WS-Addressing is used by the following advanced WebLogic JAX-WS features:

- Asynchronous client transport, as described in Chapter 4, "Invoking Web Services Asynchronously".
- WS-ReliableMessaging, as described in Chapter 6, "Using Web Services Reliable Messaging".
- Callbacks, as described in Chapter 12, "Using Callbacks".

The following sections describe how to enable WS-Addressing on the Web service or client, and explicitly define the action and fault properties.

A Note About WS-Addressing Standards Supported

WebLogic Web services support the following standards for Web service addressing:

- W3C WS-Addressing, as described at: `http://www.w3.org/2002/ws/addr/`
- Member Submission, as described at: `http://www.w3.org/Submission/ws-addressing/`

This chapter focuses on the use of W3C WS-Addressing only.

2.2 Enabling WS-Addressing on the Web Service

By default, WS-Addressing is disabled on a Web service endpoint, and any WS-Addressing headers received are ignored. You can enable WS-Addressing on the Web Service starting from Java or WSDL, as described in the following sections:

- Section 2.2.1, "Enabling WS-Addressing on the Web Service (Starting From Java)"
- Section 2.2.2, "Enabling WS-Addressing on the Web Service (Starting from WSDL)"

When you enable WS-Addressing on a Web service endpoint:

- All WS-Addressing headers are understood by the endpoint. That is, if any WS-Addressing header is received with `mustUnderstand` enabled, then no fault is thrown.
- WS-Addressing headers received are validated to ensure:
  - Correct syntax
  - Correct number of elements
  - `wsa:Action` header in the SOAP header matches what is required by the operation
- Response messages returned to the client contain the required WS-Addressing headers.

2.2.1 Enabling WS-Addressing on the Web Service (Starting From Java)

To enable WS-Addressing on the Web service starting from Java, use the `java.xml.ws.soap.Addressing` annotation on the Web service class. Optionally, you can pass one or more of the Boolean attributes defined in Table 2–2.

Table 2–2 Attributes of the @Addressing Annotation

Attribute	Description
`enabled`	Specifies whether WS-Addressing is enabled. Valid values include `true` (enabled) and `false` (disabled). This attribute defaults to `true`.
`required`	Specifies whether WS-Addressing rules are enforced for the inbound message. Valid values include `true` (enforced) and `false` (not enforced). If set to `false`, the inbound message is checked to see if WS-Addressing is enabled, and, if so, the rules are enforced. This attribute defaults to `false`.

Once enabled, the `wsaw:UsingAddressing` element is generated in the corresponding `wsdl:binding` element. For more information, see Section 2.2.2, "Enabling WS-Addressing on the Web Service (Starting from WSDL)."

The following provides an example of how to enable WS-Addressing starting from Java. In this example, WS-Addressing is enforced on the endpoint (`required` is set to `true`).

Example 2–3 Enabling WS-Addressing on the Web Service (Starting From Java)

```
package examples;
import javax.jws.WebService;
import javax.xml.ws.soap.Addressing;

@WebService(name="HelloWorld", serviceName="HelloWorldService")
@Addressing(enabled=true, required=false)

public class HelloWorld {
   public String sayHelloWorld(String message) throws MissingName { ... }
}
```

2.2.2 Enabling WS-Addressing on the Web Service (Starting from WSDL)

To enable WS-Addressing on the Web service starting from WSDL, add the `wsaw:UsingAddressing` element to the corresponding `wsdl:binding` element. Optionally, you can add the `wsdl:required` Boolean attribute to specify whether WS-Addressing rules are enforced for the inbound message. By default, this attribute is `false`.

The following provides an example of how to enable WS-Addressing starting from WSDL. In this example, WS-Addressing is enforced on the endpoint (`wsdl:required` is set to `true`).

Example 2–4 Enabling WS-Addressing on the Web Service (Starting From WSDL)

```
...
  <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
    <wsaw:UsingAddressing wsdl:required="true" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document"/>
    <operation name="sayHelloWorld">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
```

```
        <soap:body use="literal"/>
      </output>
      <fault name="MissingName">
        <soap:fault name="MissingName" use="literal"/>
      </fault>
    </operation>
  </binding>
...
```

2.3 Enabling WS-Addressing on the Web Service Client

WS-Addressing can be enabled on the Web service client implicitly or explicitly. Once enabled on the client:

- All WS-Addressing headers received on the client are understood. That is, if any WS-Addressing header is received with `mustUnderstand` enabled, then no fault is thrown.
- The JAX-WS runtime:
  - Maps all `wsaw:Action` elements, including `input`, `output`, and `fault` elements in the `wsdl:operation` to `javax.xml.ws.Action` and `javax.xml.ws.FaultAction` annotations in the generated service endpoint interface (SEI).
  - Generates `Action`, `To`, `MessageID`, and anonymous `ReplyTo` headers on the outbound request.

The following sections describe how to enable WS-Addressing on the Web service client explicitly and implicitly, and how to disable WS-Addressing explicitly.

- Section 2.3.1, "Explicitly Enabling WS-Addressing on the Web Service Client"
- Section 2.3.2, "Implicitly Enabling WS-Addressing on the Web Service Client"
- Section 2.3.3, "Disabling WS-Addressing on the Web Service Client"

2.3.1 Explicitly Enabling WS-Addressing on the Web Service Client

The Web service client can enable WS-Addressing explicitly by passing `javax.xml.ws.soap.AddressingFeature` as an argument to the `getPort` or `createDispatch` methods on the `javax.xml.ws.Service` object. Optionally, you can pass one or more of the Boolean parameters defined in Table 2–3.

Table 2–3 Parameters of the AddressingFeature Feature

Parameter	Description
`enabled`	Specifies whether WS-Addressing is enabled for an outbound message. Valid values include `true` (enabled) and `false` (disabled). This attribute defaults to `true`.
`required`	Specifies whether WS-Addressing rules are enforced for the inbound messages. Valid values include `true` (enforced) and `false` (not enforced). If set to `false`, the inbound message is checked to see if WS-Addressing is enabled, and, if so, the rules are enforced. This attribute defaults to `false`.

The following shows an example of enabling WS-Addressing on a Web service client when creating a Web service proxy, by passing the `AddressingFeature` to the `getPort` method.

Example 2–5 Enabling WS-Addressing on a Web Service Client on the Web Service Proxy

```
package examples.client;

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;
import examples.client.MissingName_Exception;
import javax.xml.ws.soap.AddressingFeature;

public class Main {
  public static void main(String[] args) throws MissingName_Exception {
    HelloWorldService service;

    try {
      service = new HelloWorldService(new URL(args[0] + "?WSDL"),
                new QName("http://examples/", "HelloWorldService") );
    } catch (MalformedURLException murl) { throw new RuntimeException(murl); }

      HelloWorld port = service.getHelloWorldPort(
        new AddressingFeature(true, true));
...
  }
}
```

The following shows an example of enabling WS-Addressing on a Web service client when creating a Dispatch instance, by passing the `AddressingFeature` to the `createDispatch` method.

Example 2–6 Enabling WS-Addressing on a Web Service Client on the Dispatch Instance

```
...
      HelloWorld port = service.getHelloWorldPort(new AddressingFeature(false));
...
```

2.3.2 Implicitly Enabling WS-Addressing on the Web Service Client

WS-Addressing is enabled implicitly if the `wsaw:UsingAddressing` extensibility element exists in the WSDL For more information, see Section 2.2.2, "Enabling WS-Addressing on the Web Service (Starting from WSDL)".

2.3.3 Disabling WS-Addressing on the Web Service Client

A Web service client may need to disable WS-Addressing processing explicitly, for example, if it has its own WS-Addressing processing module. For example, a Dispatch client in MESSAGE mode may be used to perform non-anonymous ReplyTo and FaultTo processing.

The following shows an example of how to disable explicitly WS-Addressing on a Web service client when creating a Web service proxy. In this example, the `AddressingFeature` feature is called with `enabled` set to `false`.

Example 2–7 Disabling WS-Addressing on a Web Service Client

```
...
new AddNumbersImplService().getAddNumbersImplPort(new
javax.xml.ws.AddressingFeature(false));
...
```

2.4 Associating WS-Addressing Action Properties

WS-Addressing defines an attribute, `wsaw:Action`, that can be used to explicitly associate WS-Addressing action message addressing properties with the Web service. By default, an implicit action association is made when WS-Addressing is enabled and no action is explicitly associated with the Web service.

The following sections describe how to associate WS-Addressing Action properties either explicitly or implicitly:

- Section 2.4.1, "Explicitly Associating WS-Addressing Action Properties (Starting from Java)"
- Section 2.4.2, "Explicitly Associating WS-Addressing Action Properties (Starting from WSDL)"
- Section 2.4.3, "Implicitly Associating WS-Addressing Action Properties"

2.4.1 Explicitly Associating WS-Addressing Action Properties (Starting from Java)

To explicitly associate WS-Addressing action properties with the Web service starting from Java, use the `javax.xml.ws.Action` and `javax.xml.ws.FaultAction` annotations.

Optionally, you can pass to the `@Action` annotation one or more of the attributes defined in Table 2–4.

Table 2–4 Attributes of the @Action Annotation

Attribute	Description
`input`	Associates Action message addressing property for the input message of the operation.
`output`	Associates Action message addressing property for the output message of the operation.
`fault`	Associates Action message addressing property for the fault message of the operation. Each exception that is mapped to a SOAP fault and requires explicit association must be specified using the `@FaultAction` annotation, as described in Table 2–5.

You can pass to the `@FaultAction` annotation one or more of the attributes defined in Table 2–4.

Table 2–5 Attributes of the @FaultAction Annotation

Attribute	Description
`className`	Name of the exception class. This attribute is required.
`value`	Value of the WS-Addressing Action message addressing property for the exception.

Once you explicitly associate the WS-Addressing action properties, the `wsaw:Action` attribute is generated in the corresponding `input`, `output`, and `fault` elements in the `wsdl:portType` element. For more information, see Section 2.2.2, "Enabling WS-Addressing on the Web Service (Starting from WSDL)."

The following provides an example of how to explicitly associate the WS-Addressing action message addressing properties for the input, output, and fault messages on the `sayHelloWorld` method, using the `@Action` and `@FaultAction` annotations.

Example 2–8 Example of Explicitly Associating an Action (Starting from Java)

```
...
@Action(
   input = "http://examples/HelloWorld/sayHelloWorldRequest",
   output = "http://examples/HelloWorld/sayHelloWorldResponse",
   fault = { @FaultAction(className = MissingName.class,
               value = "http://examples/MissingNameFault")})

   public String sayHelloWorld(String message) throws MissingName {
...
```

Once defined, the `wsaw:Action` element is generated in the corresponding `input`, `output`, and `fault` elements of the `wsdl:operation` element for the endpoint. For more information about these elements, see Section 2.4.2, "Explicitly Associating WS-Addressing Action Properties (Starting from WSDL)."

2.4.2 Explicitly Associating WS-Addressing Action Properties (Starting from WSDL)

To explicitly associate WS-Addressing action properties with the Web service starting from WSDL, add the `wsaw:Action` element to the corresponding `wsdl:binding` element. Optionally, you can add the `wsdl:required` Boolean attribute to specify whether WS-Addressing rules are enforced for the inbound message. By default, this attribute is `false`.

The following provides an example of how to, within the WSDL file, explicitly associate the WS-Addressing action message addressing properties for the `input`, `output`, and `fault` messages on the `sayHelloWorld` method of the HelloWorld endpoint.

Example 2–9 Example of Explicitly Associating an Action (Starting from WSDL)

```
...
  <portType name="HelloWorld">
    <operation name="sayHelloWorld">
      <input wsaw:Action="http://examples/HelloWorld/sayHelloWorldRequest"
       message="tns:sayHelloWorld"/>
      <output wsaw:Action="http://examples/HelloWorld/sayHelloWorldResponse"
       message="tns:sayHelloWorldResponse"/>
      <fault message="tns:MissingName" name="MissingName"
       wsaw:Action="http://examples/MissingNameFault"/>
    </operation>
  </portType>
...
```

2.4.3 Implicitly Associating WS-Addressing Action Properties

When WS-Addressing is enabled, if no explicit action is defined in the WSDL, the client sends an implicit `wsa:Action` header using the following formats:

- Input message action: *`targetNamespace/portTypeName/inputName`*
- Output message action: *`targetNamespace/portTypeName/outputName`*

- Fault message action:
  targetNamespace/portTypeName/operationName/Fault/*FaultName*

```
targetNamespace/portTypeName/[inputName | outputName]
```

For example, for the following WSDL excerpt:

```
<definitions targetNamespace="http://examples/"...>
...
  <portType name="HelloWorld">
    <operation name="sayHelloWorld">
      <input message="tns:sayHelloWorld" name="sayHelloRequest"/>
      <output message="tns:sayHelloWorldResponse" name="sayHelloResponse"/>
      <fault message="tns:MissingName" name="MissingName" />
    </operation>
  </portType>
...
</definitions>
```

The default input and output actions would be defined as follows:

- Input message action: `http://examples/HelloWorld/sayHelloRequest`
- Output message action: `http://examples/HelloWorld/sayHelloResponse`
- Fault message action:
  `http://examples/HelloWorld/sayHelloWorld/Fault/MissingName`

If the input or output message name is not specified in the WSDL, the operation name is used with Request or Response appended, respectively. For example: `sayHelloWorldRequest` or `sayHelloWorldResponse`.

2.5 Configuring Anonymous WS-Addressing

In some cases, the network technologies used in an environment (such as, firewalls, Dynamic Host Configuration Protocol (DHCP), and so on) may prohibit the use of globally addressed URI for a given endpoint. To enable non-addressable endpoints to exchange messages, the WS-Addressing specification supports "anonymous" endpoints using the `wsaw:Anonymous` element.

The `wsaw:Anonymous` element can be set to one of the values defined in Table 2–6.

Table 2–6 Valid Values for the wsaw:Anonymous Element

Value	Description
`optional`	The response endpoint EPR in a request message may contain an anonymous URI.
`required`	The response endpoint EPR in a request message must contain an anonymous URL. Otherwise, an `InvalidAddressingHeader` fault is returned.
`prohibited`	The response endpoint EPRs in a request message must not contain an anonymous URI. Otherwise, an `InvalidAddressingHeader` fault is returned.

To configure anonymous WS-Addressing:

1. Enable WS-Addressing on the Web service, add the `wsaw:UsingAddressing` element to the corresponding `wsdl:binding` element. For more information, see

Section 2.2.2, "Enabling WS-Addressing on the Web Service (Starting from WSDL)."

Note: When anonymous WS-Addressing is enabled, the `wsdl:required` attribute must not be enabled in the `wsaw:UsingAddressing` element.

---

2. Add the `wsaw:Anonymous` element to the `wsdl:operation` within the `wsdl:binding` element.

The following provides an example of how to enable anonymous WS-Addressing in the WSDL file. In this example, anonymous addressing is required.

Example 2–10 Enabling Anonymous WS-Addressing on the Web Service

```
...
  <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
    <wsaw:UsingAddressing wsdl:required="true" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document"/>
    <operation name="sayHelloWorld">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
      <fault name="MissingName">
        <soap:fault name="MissingName" use="literal"/>
      </fault>
    </operation>
    <wsaw:Anonymous>required</wsaw:Anonymous>
  </binding>
...
```

3

Roadmaps for Developing Web Service Clients

This chapter presents best practices for developing WebLogic Web service clients for Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

- Section 3.1, "Roadmap for Developing Web Service Clients"
- Section 3.2, "Roadmap for Developing Asynchronous Web Service Clients"

Note: It is assumed that you are familiar with the general concepts for developing Web service clients, as described "Invoking Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

For best practices for developing reliable Web service clients, see Chapter 5, "Roadmap for Developing Reliable Web Services and Clients."

In the following sections, *client instance* can be a port or a Dispatch instance.

3.1 Roadmap for Developing Web Service Clients

Table 3.1 provides best practices for developing Web service clients, including an example that illustrates the best practices presented. For additional best practices when developing asynchronous Web service clients, see Section 3.2, "Roadmap for Developing Asynchronous Web Service Clients".

Table 3–1 Roadmap for Developing Web Service Clients

Best Practice	Description
Synchronize use of client instances.	Create client instances as you need them; do not store them long term.
Use a stored list of features, including client ID, to create client instances.	Define all features for the Web service client instance, including client ID, so that they are consistent each time the client instance is created. For example: `_service.getBackendServicePort(_features);`
Explicitly define the client ID.	Use the `ClientIdentityFeature` to define the client ID explicitly. This client ID is used to group statistics and other monitoring information, and for reporting runtime validations, and so on. For more information, see "Managing Client Identity" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*. **Note:** Oracle strongly recommends that you define the client ID explicitly. If not explicitly defined, the server generates the client ID automatically, which may not be user-friendly.
Explicitly close client instances when processing is complete.	For example: `((java.io.Closeable)port).close();` If not closed explicitly, the client instance will be closed automatically when it goes out of scope. **Note**: The client ID remains registered and visible until the container (Web application or EJB) is deactivated. For more information, see "Client Identity Lifecycle" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

The following example illustrates best practices for developing Web service clients.

Example 3–1 Web Service Client Best Practices Example

```
import java.io.IOException;
import java.util.*;

import javax.servlet.*;
import javax.xml.ws.*;

import weblogic.jws.jaxws.client.ClientIdentityFeature;

/**
 * Example client for invoking a Web service.
 */
public class BestPracticeClient
  extends GenericServlet {

  private BackendServiceService _service;
  private WebServiceFeature[] _features;
  private ClientIdentityFeature _clientIdFeature;

  @Override
  public void init()
    throws ServletException {

    // Create a single instance of a Web service as it is expensive to create repeatedly.
    if (_service == null) {
      _service = new BackendServiceService();
    }

    // Best Practice: Use a stored list of features, per client ID, to create client instances.
    // Define all features for the Web service client instance, per client ID, so that they are
```

```
  // consistent each time the client instance is created. For example:
  // _service.getBackendServicePort(_features);

  List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

  // Best Practice: Explicitly define the client ID.
  // TODO: Maybe allow ClientIdentityFeature to store other features, and
  //       then create new client instances simply by passing the
  //       ClientIdentityFeature (and the registered features are used).
  _clientIdFeature = new ClientIdentityFeature("MyBackendServiceClient");
  features.add(_clientIdFeature);

  // Set the features used when creating clients with
  // the client ID "MyBackendServiceClient". The features are stored in an array to
  // reinforce that the list should be treated as immutable.
  _features = features.toArray(new WebServiceFeature[features.size()]);
}

@Override
public void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException {

  // ... Read the servlet request ...

  // Best Practice: Synchronize use of client instances.
  // Create a Web service client instance to talk to the backend service.
  // Note, at this point the client ID is 'registered' and becomes
  // visible to monitoring tools such as the Administration Console and WLST.
  // The client ID *remains* registered and visible until the container
  // (the Web application hosting our servlet) is deactivated (undeployed).
  //
  // A client ID can be used when creating multiple client instances (port or Dispatch client).
  // The client instance should be created with the same set of features each time, and should
  // use the same service class and refer to the same port type.
  // A given a client ID should be used for a given port type, but not across port types.
  // It can be used for both port and Dispatch clients.
  BackendService port =
    _service.getBackendServicePort(_features);

  // Set the endpoint address for BackendService.
  ((BindingProvider)port).getRequestContext().
    put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        "http://localhost:7001/BestPracticeService/BackendService");

  // Print out the explicit client ID, and compare it to the client ID
  // that would have been generated automatically for the client instance.
  showClientIdentity();

  // Make the invocation on our real port
  String request = "Make a cake";
  System.out.println("Invoking DoSomething with request: " + request);
  String response = port.doSomething(request);
  System.out.println("Got response: " + response);
  res.getWriter().write(response);

  // Best Practice: Explicitly close client instances when processing is complete.
  // If not closed, the client instance will be closed automatically when it goes out of
  // scope. Note, this client ID will remain registered and visible until our
  // container (Web application) is undeployed.
  ((java.io.Closeable)port).close();
```

```
  }

  /**
    // Print out the client's full ID, which is a combination of
    // the client ID provided above and qualifiers from the application and
    // Web application that contain the client. Then compare this with the client ID that
    // would have been generated for the client instance if not explicitly set.
    //
  private void showClientIdentity()
    throws IOException {

    System.out.println("Client Identity is: " + _clientIdFeature.getClientId());

    // Create a client instance without explicitly defining the client ID to view the
    // client ID that is generated automatically.
    ClientIdentityFeature dummyClientIdFeature =
      new ClientIdentityFeature(null);
    BackendService dummyPort =
      _service.getBackendServicePort(dummyClientIdFeature);
    System.out.println("Generated Client Identity is: " +
                       dummyClientIdFeature.getClientId());
    // Best Practice: Explicitly close client instances when processing is complete.
    // If not closed, the client instance will be closed automatically when it goes out of
    // scope. Note, this client ID will remain registered and visible until our
    // container (Web application) is undeployed.
    ((java.io.Closeable)dummyPort).close();
  }

  @Override
  public void destroy() {
  }
}
```

3.2 Roadmap for Developing Asynchronous Web Service Clients

Table 3.2 provides best practices for developing asynchronous Web service clients, including an example that illustrates the best practices presented. These guidelines should be used in conjunction with the general guidelines provided in Section 3.1, "Roadmap for Developing Web Service Clients".

Table 3–2 Roadmap for Developing Asynchronous Web Service Clients

Best Practice	Description
Define a port-based asynchronous callback handler, `AsyncClientHandlerFeature`, for asynchronous and dispatch callback handling.	Use of `AsyncClientHandlerFeature` is recommended as a best practice when using asynchronous invocation due to its scalability and ability to survive a JVM restart. It can be used by any client (survivable or not.) For information, see Section 4.5.2, "Developing the Asynchronous Handler Interface".
Define a singleton port instance and initialize it when the client container initializes (upon deployment).	Creation of the singleton port: ▪ Triggers the asynchronous response endpoint to be published upon deployment. ▪ Supports failure recovery by re-initializing the singleton port instance after VM restart. Within a cluster, initialization of a singleton port will ensure that all member servers in the cluster publish an asynchronous response endpoint.This ensures that the asynchronous response messages can be delivered to any member server and optionally forwarded to the correct server via in-place cluster routing. For complete details, see Section 4.10, "Clustering Considerations for Asynchronous Web Service Messaging.".
If using Make Connection for clients behind a firewall, set the Make Connection polling interval to a value that is realistic for your scenario.	The Make Connection polling interval should be set as high as possible to avoid unnecessary polling overhead, but also low enough to allow responses to be retrieved in a timely fashion. A recommended value for the Make Connection polling interval is one-half of the expected average response time of the Web service being invoked. For more information setting the Make Connection polling interval, see Section 4.6.2.2, "Configuring the Polling Interval." **Note**: This best practice is not demonstrated in Example 3–2.
If using the JAX-WS Reference Implementation (RI), implement the `AsyncHandler<T>` interface.	Use of the `AsyncHandler<T>` interface is more efficient than the `Response<T>` interface. For more information and an example, see Section 4.7, "Using the JAX-WS Reference Implementation". **Note**: This best practice is not demonstrated in Example 3–2.
Define a Work Manager and set the thread pool minimum size constraint (`min-threads-constraint`) to a value that is at least as large as the expected number of concurrent requests or responses into the service.	For example, if a Web service client issues 20 requests in rapid succession, the recommended thread pool minimum size constraint value would be 20 for the application hosting the client. If the configured constraint value is too small, performance can be severely degraded as incoming work waits for a free processing thread. For more information about the thread pool minimum size constraint, see "Constraints" in *Configuring Server Environments for Oracle WebLogic Server*.

The following example illustrates best practices for developing asynchronous Web service clients.

Example 3–2 Asynchronous Web Service Client Best Practices Example

```
import java.io.*;
import java.util.*;

import javax.servlet.*
import javax.xml.ws.*

import weblogic.jws.jaxws.client.ClientIdentityFeature;
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
import weblogic.jws.jaxws.client.async.AsyncClientTransportFeature;

import com.sun.xml.ws.developer.JAXWSProperties;
```

```
/**
 * Example client for invoking a Web service asynchronously.
 */
public class BestPracticeAsyncClient
  extends GenericServlet {

  private static final String MY_PROPERTY = "MyProperty";

  private BackendServiceService _service;
  private WebServiceFeature[] _features;
  private BackendService _singletonPort;

  private static String _lastResponse;
  private static int _requestCount;

  @Override
  public void init()
    throws ServletException {

    // Only create the Web service object once as it is expensive to create repeatedly.
    if (_service == null) {
      _service = new BackendServiceService();
    }

    // Best Practice: Use a stored list of features, including client ID, to create client
    // instances.
    // Define all features for the Web service client instance, including client ID, so that they
    // are consistent each time the client instance is created. For example:
    // _service.getBackendServicePort(_features);

    List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

    // Best Practice: Explicitly define the client ID.
    ClientIdentityFeature clientIdFeature =
      new ClientIdentityFeature("MyBackendServiceAsyncClient");
    features.add(clientIdFeature);

    // Asynchronous endpoint
    AsyncClientTransportFeature asyncFeature =
      new AsyncClientTransportFeature(getServletContext());
    features.add(asyncFeature);

    // Best Practice: Define a port-based asynchronous callback handler,
    // AsyncClientHandlerFeature, for asynchronous and dispatch callback handling.
    BackendServiceAsyncHandler handler =
      new BackendServiceAsyncHandler() {
        // This class is stateless and should not depend on
        // having member variables to work with across restarts.
        public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
          // ... Handle Response ...
          try {
            DoSomethingResponse response = res.get();
            res.getContext();
            _lastResponse = response.getReturn();
            System.out.println("Got async response: " + _lastResponse);
            // Retrieve the request property. This property can be used to
            // 'remember' the context of the request and subsequently process
            // the response.
            Map<String, Serializable> requestProps =
              (Map<String, Serializable>)
```

```
            res.getContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
          String myProperty = (String)requestProps.get(MY_PROPERTY);
          System.out.println("Got MyProperty value propagated from request: "+
                             myProperty);
        } catch (Exception e) {
          _lastResponse = e.toString();
          e.printStackTrace();
        }
      }
    };
  AsyncClientHandlerFeature handlerFeature =
    new AsyncClientHandlerFeature(handler);
  features.add(handlerFeature);

  // Set the features used when creating clients with
  // the client ID "MyBackendServiceAsyncClient".

  _features = features.toArray(new WebServiceFeature[features.size()]);

  // Best Practice: Define a singleton port instance and initialize it when
  // the client container initializes (upon deployment).
  // The singleton port will be available for the life of the servlet.
  // Creation of the singleton port triggers the asynchronous response endpoint to be published
  // and it will remain published until our container (Web application) is undeployed.
  // Note, the destroy() method will be called before this.
  // The singleton port ensures proper/robust operation in both
  // recovery and clustered scenarios.
  _singletonPort = _service.getBackendServicePort(_features);
}

@Override
public void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException {

  // TODO: ... Read the servlet request ...

  // For this simple example, echo the _lastResponse captured from
  // an asynchronous DoSomethingResponse response message.

  if (_lastResponse != null) {
    res.getWriter().write(_lastResponse);
    _lastResponse = null; // Clear the response so we can get another
    return;
  }

  // Set _lastResponse to NULL to to support the invocation against
  // BackendService to generate a new response.

  // Best Practice: Synchronize use of client instances.
  // Create another client instance using the *exact* same features used when creating _
  // singletonPort. Note, this port uses the same client ID as the singleton port
  // and it is effectively the same as the singleton
  // from the perspective of the Web services runtime.
  // This port will use the asynchronous response endpoint for the client ID,
  // as it is defined in the _features list.
  BackendService anotherPort =
    _service.getBackendServicePort(_features);

  // Set the endpoint address for BackendService.
  ((BindingProvider)anotherPort).getRequestContext().
```

```
      put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
          "http://localhost:7001/BestPracticeService/BackendService");

    // Add a persistent context property that will be retrieved on the
    // response. This property can be used as a reminder of the context of this
    // request and subsequently process the response. This property will *not*
    // be passed over the wire, so the properties can change independent of the
    // application message.
    Map<String, Serializable> persistentContext =
      (Map<String, Serializable>)((BindingProvider)anotherPort).
        getRequestContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
    String myProperty = "Request " + (++_requestCount);
    persistentContext.put(MY_PROPERTY, myProperty);
    System.out.println("Request being made with MyProperty value: " +
                       myProperty);

    // Make the asychronous invocation. The asynchronous handler implementation (set
    // into the AsyncClientHandlerFeature above) receives the response.
    String request = "Dance and sing";
    System.out.println("Invoking DoSomething asynchronously with request: " +
                       request);
    anotherPort.doSomethingAsync(request);

    // Return a canned string indicating the response was not received
    // synchronously. Client will need to invoke the servlet again to get
    // the response.
    res.getWriter().write("Waiting for response...");

    // Best Practice: Explicitly close client instances when processing is complete.
    // If not closed explicitly, the port will be closed automatically when it goes out of scope.
    ((java.io.Closeable)anotherPort).close();
  }

  @Override
  public void destroy() {

    try {
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the singleton port created during initialization. Note, the asynchronous
      // response endpoint generated by creating _singletonPort *remains*
      // published until our container (Web application) is undeployed.
      ((java.io.Closeable)_singletonPort).close();

      // Upon return, the Web application is undeployed, and the asynchronous
      // response endpoint is stopped (unpublished). At this point,
      // the client ID used for _singletonPort will be unregistered and will no longer be
      // visible from the Administration Console and WLST.
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

4

Invoking Web Services Asynchronously

This chapter describes how to invoke asynchronously WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

- Section 4.1, "Overview of Asynchronous Web Service Invocation"
- Section 4.2, "Steps to Invoke Web Services Asynchronously"
- Section 4.3, "Configuring Your Servers for Asynchronous Web Service Invocation"
- Section 4.4, "Building the Client Artifacts for Asynchronous Web Service Invocation"
- Section 4.5, "Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)"
- Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)"
- Section 4.7, "Using the JAX-WS Reference Implementation"
- Section 4.8, "Propagating Request Context to the Response"
- Section 4.9, "Monitoring Asynchronous Web Service Invocation"
- Section 4.10, "Clustering Considerations for Asynchronous Web Service Messaging"

> **Note:** See also Section 3, "Roadmaps for Developing Web Service Clients".

4.1 Overview of Asynchronous Web Service Invocation

To support asynchronous Web services invocation, WebLogic Web services can use an asynchronous client programming model, asynchronous transport, or both.

Table 4–1 provides a description and key benefits of the asynchronous client programming model and transport types, and introduces the configuration options available to support asynchronous Web service invocation.

> **Note:** The method of generating a WSDL for the asynchronous Web service containing two one-way operations defined as two portTypes—one for the asynchronous operation and one for the callback operation—is not supported in the current release.

Table 4–1 Support for Asynchronous Web Service Invocation

Type	Description	Benefits
Client programming model	Describes the invocation semantics used to call a Web service operation: synchronous or asynchronous. When you invoke a Web service *synchronously*, the invoking client application waits for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the Web service might be adequate. However, because request processing can be delayed, it is often useful for the client application to continue its work and handle the response later on. By calling a Web service *asynchronously*, the client can continue its processing, without interruption, and be notified when the asynchronous response is returned. To support asynchronous invocation, you generate automatically an asynchronous flavor of each operation on a Web service port using the `clientgen` Ant task, as described later in Section 4.4, "Building the Client Artifacts for Asynchronous Web Service Invocation." Then, you add methods in your client, including your business logic, that handle the asynchronous response or failures when it returns later on. Finally, to invoke a Web service asynchronously, rather than invoking the operation directly, you invoke the asynchronous flavor of the operation. For example, rather than invoking an operation called `addNumbers` directly, you would invoke `addNumbersAsync` instead.	Asynchronous invocation enables Web service clients to initiate a request to a Web service, continue processing without blocking, and receive the response at some point in the future.
Transport	There are three transport types: asynchronous client transport, Make Connection transport, and synchronous transport. For a comparison of each transport type, see Table 4–2.	Asynchronous client transport and Make Connection transport deliver the following key benefits: ■ Improves fault tolerance in the event of network outages. ■ Enables servers to absorb more efficiently spikes in traffic.
Configuration	Configure Web service persistence and buffering (optional) to support asynchronous Web service invocation. For more information, see Section 4.3, "Configuring Your Servers for Asynchronous Web Service Invocation."	Benefits of configuring the Web service features include: ■ Persistence supports long running requests and provides the ability to survive server restarts. ■ Buffering enables all requests to a Web service to be handled asynchronously.

Table 4–2 summarizes the transport types that WebLogic Server supports for invoking a Web service asynchronously (or synchronously, if configured) from a Web service client.

Table 4–2 Transport Types for Invoking Web Services Asynchronously

Transport Types	Description
Asynchronous Client Transport	Provides a scalable asynchronous client programming model through the use of an *addressable* client-side asynchronous response endpoint and WS-Addressing. Asynchronous client transport decouples the delivery of the response message from the initiating transport request used to send the request message. The response message is sent to the asynchronous response endpoint using a new connection originating from the Web service. The client correlates request and response messages through WS-Addressing headers. Asynchronous client transport provides improved fault tolerance and enables servers to better absorb spikes in server load. For details about using asynchronous client transport, see Section 4.5, "Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)." Asynchronous client transport supports the following programming models: ■ Asynchronous and dispatch callback handling using one of the following methods: - Port-based asynchronous callback handler, `AsyncClientHandlerFeature`, described in Section 4.5.2, "Developing the Asynchronous Handler Interface." This is recommended as a **best practice** when using asynchronous invocation due to its scalability and ability to survive a JVM restart. - Per-request asynchronous callback handler, as described in Section 4.7, "Using the JAX-WS Reference Implementation." ■ Asynchronous polling, as described in Section 4.7, "Using the JAX-WS Reference Implementation." ■ Synchronous invocation by enabling a flag, as described in Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations."

Table 4–2 (Cont.) Transport Types for Invoking Web Services Asynchronously

Transport Types	Description
Make Connection Transport	Enables asynchronous Web service invocation from behind a firewall using Web Services Make Connection 1.1 or 1.0. Make Connection is a client polling mechanism that provides an alternative to asynchronous client transport. As with asynchronous client transport, Make Connection enables the decoupling of the response message from the initiating transport request used to send the request message. However, unlike asynchronous client transport which requires an addressable asynchronous response endpoint to forward the response to, with Make Connection typically the sender of the request message is *non-addressable* and unable to accept an incoming connection. For example, when the sender is located behind a firewall. Make Connection transport provides improved fault tolerance and enables servers to better absorb spikes in server load. For details about Make Connection transport, see Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)." Make Connection transport is recommended as a best practice when using asynchronous invocation from behind a firewall due to its scalability and ability to survive a JVM restart. It supports the following programming models: ■ Asynchronous and dispatch callback handling using one of the following methods: - Port-based asynchronous callback handler, `AsyncClientHandlerFeature`, described in Section 4.5.2, "Developing the Asynchronous Handler Interface". - Per-request asynchronous callback handler, as described in Section 4.7, "Using the JAX-WS Reference Implementation" ■ Asynchronous polling, as described in Section 4.7, "Using the JAX-WS Reference Implementation". ■ Synchronous invocation by enabling a flag, as described in Section 4.6.2.4, "Configuring Make Connection as the Transport for Synchronous Methods". Use of Make Connection transport with `AsyncClientHandlerFeature` is recommended as a best practice when using asynchronous invocation due to its scalability and ability to survive a JVM restart.
Synchronous Transport	Provides support for synchronous and asynchronous Web service invocation with very limited support for WS-Addressing. For details, see Section 4.7, "Using the JAX-WS Reference Implementation". Synchronous transport is recommended when using synchronous invocation. It can be used for asynchronous invocation, as well, though this is not considered a best practice. It supports the following programming models: ■ Asynchronous and dispatch callback handling on a per request basis using the standard JAX-WS RI implementation, described in Section 4.7, "Using the JAX-WS Reference Implementation". ■ Asynchronous polling, as described in Section 4.7, "Using the JAX-WS Reference Implementation". ■ Synchronous invocation.

4.2 Steps to Invoke Web Services Asynchronously

This section describes the steps required to invoke Web services asynchronously.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the Web services. For more information, see *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 4–3 Steps to Invoke Web Services Asynchronously

#	Step	Description
1	Configure Web service persistence to support asynchronous Web service invocation.	Configure Web service persistence on the servers hosting the Web service and client to retain context information required for processing a message at the Web service or client. For more information, see Section 4.3, "Configuring Your Servers for Asynchronous Web Service Invocation". **Note**: This step is not required if you are programming the Web service client using the standard JAX-WS RI implementation and synchronous transport (in Step 3), as described in Section 4.7, "Using the JAX-WS Reference Implementation".
2	Configure Web service buffering to enable the Web service to process requests asynchronously. (Optional)	This step is optional. To configure the Web service to process requests asynchronously, configure buffering on the server hosting the Web service. Buffering enables you to store messages in a JMS queue for asynchronous processing by the Web service. For more information, see Section 4.3, "Configuring Your Servers for Asynchronous Web Service Invocation".
3	Build the client artifacts required for asynchronous invocation.	To generate asynchronous polling and asynchronous callback handler methods in the service endpoint interface, create an external binding declarations that enables asynchronous mappings and pass the bindings file as an argument to the `clientgen` when compiling the client. See Section 4.4, "Building the Client Artifacts for Asynchronous Web Service Invocation".
4	Implement the Web service client based on the transport and programming model required.	Refer to one of the following sections based on the transport and programming model required: ■ Use asynchronous client transport, as described in Section 4.5, "Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)". (Recommended as a best practice.) ■ Enable asynchronous access from behind a firewall using Make Connection. See Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)". ■ Implement standard JAX-WS programming models, such as asynchronous polling or per-request asynchronous callback handling, using synchronous transport. See Section 4.7, "Using the JAX-WS Reference Implementation". When using Web services in a cluster, review the guidelines described in Section 4.10, "Clustering Considerations for Asynchronous Web Service Messaging".
5	Compile the Web service client and package the client artifacts.	For more information, see "Compiling and Running the Client Application" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 4–3 (Cont.) Steps to Invoke Web Services Asynchronously

#	Step	Description
6	Deploy the Web service client.	See "Deploying and Undeploying WebLogic Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
7	Monitor the Web service client.	You can monitor runtime information for clients that invoke Web services asynchronously, such as number of invocations, errors, faults, and so on, using the Administration Console or WLST. See Section 4.9, "Monitoring Asynchronous Web Service Invocation".

4.3 Configuring Your Servers for Asynchronous Web Service Invocation

Note: This step is not required if you are programming the Web service client using the standard JAX-WS RI implementation and synchronous transport, as described in Section 4.7, "Using the JAX-WS Reference Implementation".

To support asynchronous Web service invocation, you need to configure the features defined in the following table on the servers to which the Web service and client are deployed.

Table 4–4 Configuration for Asynchronous Web Service Invocation

Feature	Description
Persistence	Web service persistence is used to save the following types of information: ■ Client identity and properties ■ SOAP message, including its headers and body ■ Context properties required for processing the message at the Web service or client (for both asynchronous and synchronous messages) The Make Connection transport protocol makes use of Web service persistence as follows: ■ Web service persistence configured on the MC Receiver (Web service) persists response messages that are awaiting incoming Make Connection messages for the Make Connection anonymous URI to which they are destined. Messages are persisted until either they are returned as a response to an incoming Make Connection message or the message reaches the maximum lifetime for a persistent store object, resulting in the message being cleaned from the store. ■ Web service persistence configured on the MC Initiator (Web service client) is used with the asynchronous client handler feature to recover after a VM restart. You can configure Web service persistence using the Configuration Wizard to extend the WebLogic Server domain using a Web services-specific extension template. Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Administration Console or WLST. For information about configuring Web service persistence, see Section 7.3.3, "Configuring Web Service Persistence for Web Service Clients". For information about the APIs available for persisting client and message information, see Section 4.8, "Propagating Request Context to the Response".

Table 4–4 (Cont.) Configuration for Asynchronous Web Service Invocation

Feature	Description
Message buffering	When a buffered operation is invoked by a client, the request is stored in a JMS queue and WebLogic Server processes it asynchronously. If WebLogic Server goes down while the request is still in the queue, it will be processed as soon as WebLogic Server is restarted. Message buffering is configured on the server hosting the Web service. For configuration information, see Chapter 8, "Configuring Message Buffering for Web Services". **Note**: Message buffering is enabled automatically on the Web service client.

4.4 Building the Client Artifacts for Asynchronous Web Service Invocation

Using the WebLogic Server client-side tooling (for example, `clientgen`), you can generate automatically the client artifacts required for asynchronous Web service invocation. Specifically, the following artifacts are generated:

- Service endpoint interfaces for invoking the Web service asynchronously with or without a per-request asynchronous callback handler. For example, if the Web service defined the following method:

  ```
  public int addNumbers(int opA, int opB) throws MyException
  ```

  Then the following methods will be generated:

  ```
  public Future<?> addNumbersAsync(int opA, int opB,
                   AsyncHandler<AddNumbersResponse>)
  public Response<AddNumbersResponse> addNumbersAsync(int opA, int opB)
  ```

- Asynchronous handler interface for implementing a handler and setting it on the port using `AsyncClientHandlerFeature`. The asynchronous handler interface is named as follows: *`portInterfaceName`*`AsyncHandler`, where *`portInterfaceName`* specifies the name of the port interface.

  For example, for a Web service with a port type name `AddNumbersPortType`, an asynchronous handler interface named `AddNumbersPortTypeAsyncHandler` is generated with the following method:

  ```
  public void onAddNumbersResponse(Response<AddNumbersResponse>)
  ```

  The `AsyncClientHandlerFeature` is described later, in Section 4.5.2, "Developing the Asynchronous Handler Interface".

To generate asynchronous client artifacts in the service endpoint interface when the WSDL is compiled, enable the `jaxws:enableAsyncMapping` binding declaration in the WSDL file.

Alternatively, you can create an external binding declarations file that contains all binding declarations for a specific WSDL or XML Schema document. Then, pass the binding declarations file to the `<binding>` child element of the `wsdlc`, `jwsc`, or `clientgen` Ant task. For more information, see "Creating an External Binding Declarations File Using JAX-WS Binding Declarations" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

The following provides an example of a binding declarations file (`jaxws-binding.xml`) that enables the `jaxws:enableAsyncMapping` binding declaration:

```
<bindings
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="AddNumbers.wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
        <package name="examples.webservices.async"/>
        <enableAsyncMapping>true</enableAsyncMapping>
    </bindings>
</bindings>
```

Then, to update the `build.xml` file to generate client artifacts necessary to invoke a Web service operation asynchronously:

1. Use the `taskdef` Ant task to define the full classname of the `clientgen` Ant tasks.
2. Add a target that includes a reference to the external binding declarations file containing the asynchronous binding declaration, as defined above. In this case, the `clientgen` Ant task generates both synchronous and asynchronous flavors of the Web service operations in the JAX-WS stubs.

For example:

```
<taskdef name="clientgen"
    classname="weblogic.wsee.tools.anttasks.ClientGenTask" />

<target name="build_client">

<clientgen
      type="JAXWS"
      wsdl="AddNumbers.wsdl"
      destDir="${clientclasses.dir}"
      packageName="examples.webservices.async.client">
      <binding file="jaxws-binding.xml" />
    </clientgen>
    <javac
      srcdir="${clientclass-dir}" destdir="${clientclass-dir}"
      includes="**/*.java"/>
    <javac
      srcdir="src" destdir="${clientclass-dir}"
      includes="examples/webservices/async/client/**/*.java"/>

</target>
```

4.5 Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)

The asynchronous client transport feature provides a scalable asynchronous client programming model. Specifically, this feature:

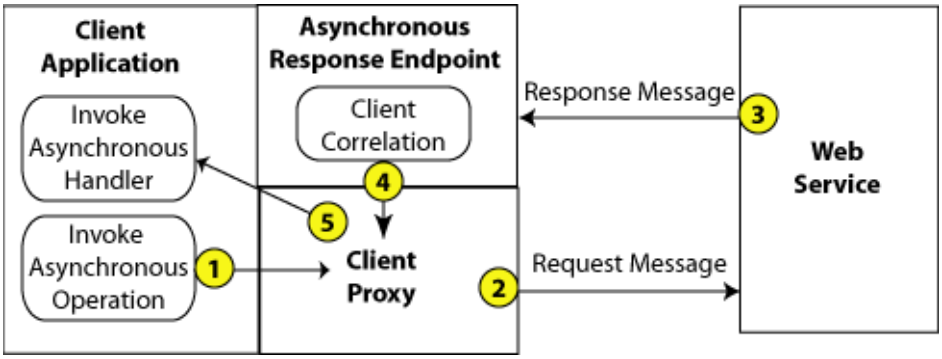
- Publishes a client-side asynchronous response endpoint, shown in Figure 4–1.
- Creates and publishes a service implementation that invokes the requested asynchronous handler implementation.
- Automatically adds WS-Addressing non-anonymous ReplyTo headers to all non-one-way, outbound messages. This header references the published response endpoint.

- Correlates asynchronous request and response messages using the facilities listed above.

When the asynchronous client transport feature is enabled, all other JAX-WS client programming models (such as asynchronous polling, callback handler, dispatch, and so on) continue to be supported. Synchronous Web service operations will, by default, use synchronous transport, unless explicitly configured to use asynchronous client transport feature when enabling the feature.

The following figure shows the message flow used by the asynchronous client transport feature.

Figure 4–1 Asynchronous Client Transport Feature



As shown in the previous figure:

1. The client enables the asynchronous client transport feature on the client proxy and invokes an asynchronous Web service operation.
2. The Web service operation is invoked via the client proxy.
3. The Web service processes the request and sends a response message (at some time in the future) back to the client. The response message is sent to the client's asynchronous response endpoint. The address of the asynchronous response endpoint is maintained in the WS-Addressing headers.
4. The response message is forwarded to the appropriate client via the client proxy.
5. The client asynchronous handler is invoked to handle the response message.

The following sections describe how to develop scalable asynchronous JAX-WS clients using asynchronous client transport:

- Section 4.5.1, "Enabling and Configuring the Asynchronous Client Transport Feature"
- Section 4.5.2, "Developing the Asynchronous Handler Interface"
- Section 4.5.3, "Propagating User-defined Request Context to the Response"

4.5.1 Enabling and Configuring the Asynchronous Client Transport Feature

> **Note:** The Make Connection and asynchronous client transport features are mutually exclusive. If you attempt to enable both features on the same Web service client, an error is returned. For more information about Make Connection, see Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)".

To enable the asynchronous client transport feature on a client, pass an instance of `weblogic.jws.jaxws.client.async.AsyncClientTransportFeature` as a parameter when creating the Web service proxy or dispatch. The following sections describe how to enable and configure the asynchronous client transport feature on a client:

- Section 4.5.1.1, "Configuring the Address of the Asynchronous Response Endpoint"
- Section 4.5.1.2, "Configuring the ReplyTo and FaultTo Headers of the Asynchronous Response Endpoint"
- Section 4.5.1.3, "Configuring the Context Path of the Asynchronous Response Endpoint"
- Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint"
- Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations"

The asynchronous response endpoint described by the `AsyncClientTransportFeature` is used by all client instances that share the same client ID and is in effect from the time the first client instance using the client ID is published. The asynchronous response endpoint remains published until the client ID is explicitly disposed or the container for the client is deactivated (for example, the host Web application or EJB is undeployed). For more information about managing the client ID, see "Managing Client Identity" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

The asynchronous response endpoint address is generated automatically using the following format:

`http://`*`contextAddress`*`:`*`port`*`/`*`context`*`/`*`targetPort`*`-AsyncResponse`

In the above:

- *`contextAddress`*`:`*`port`*—Specifies one of the following:
  - If clustered application, cluster address and port.
  - If not clustered application, default WebLogic Server address and port for the selected protocol.
  - If no default address is defined, first network channel address for the given protocol. For more information about network channels, see "Configuring Network Resources" in *Configuring Server Environments for Oracle WebLogic Server*.
- *`context`*—Current servlet context, if running within an existing context. Otherwise, a new context named by the UUID and scoped to the application.

- *targetPort*-AsyncResponse—Port name of the service accessed by the client appended by -AsyncResponse.

You can configure the asynchronous client transport feature, as described in the following sections:

- Section 4.5.1.1, "Configuring the Address of the Asynchronous Response Endpoint"
- Section 4.5.1.2, "Configuring the ReplyTo and FaultTo Headers of the Asynchronous Response Endpoint"
- Section 4.5.1.3, "Configuring the Context Path of the Asynchronous Response Endpoint"
- Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint"
- Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations"

For more information about the AsyncClientTransportFeature() constructor formats, see the *WebLogic Server Javadoc*.

4.5.1.1 Configuring the Address of the Asynchronous Response Endpoint

You can configure an address for the asynchronous response endpoint by passing it as an argument to the AsyncClientTransportFeature, as follows:

```
String responseAddress = "http://myserver.com:7001/myReliableService/myClientCallback";
AsyncClientTransportFeature asyncFeature = new AsyncClientTransportFeature(responseAddress);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

The specified address must be a legal address for the server or cluster (including the network channels or proxy addresses). Ephemeral ports are not supported. The specified context must be scoped within the current application or refer to an unused context; it cannot refer to a context that is scoped to another deployed application, otherwise an error is thrown.

The following tables summarizes the constructors that can be used to configure the address of the asynchronous response endpoint.

Table 4–5 Constructors for Configuring the Address of the Asynchronous Response Endpoint

Constructor	Description
AsyncClientTransportFeature(java.lang.String *address*)	Configures the address of the asynchronous response endpoint.
AsyncClientTransportFeature(java.lang.String *address*, boolean *doPublish*)	Configures the following: ■ Address of the asynchronous response endpoint. ■ Whether to publish the endpoint at the specified address. For more information, see Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint".
AsyncClientTransportFeature(java.lang.String *address*, boolean *doPublish*, boolean *useAsyncWithSyncInvoke*)	Configures the following: ■ Address of the asynchronous response endpoint. ■ Whether to publish the endpoint at the specified address. For more information, see Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint". ■ Whether to enable asynchronous client transport for synchronous operations. For more information, see Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations".

4.5.1.2 Configuring the ReplyTo and FaultTo Headers of the Asynchronous Response Endpoint

You can configure the address to use for all outgoing ReplyTo and FaultTo headers of type `javax.xml.ws.wsaddressing.W3CEndpointReference` for the asynchronous response endpoint by passing them as arguments to the `AsyncClientTransportFeature`.

For example, to configure only the ReplyTo header address:

```
W3CEndpointReference replyToAddress =
"http://myserver.com:7001/myReliableService/myClientCallback";
AsyncClientTransportFeature asyncFeature = new AsyncClientTransportFeature(replyToAddress);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

To configure both the ReplyTo and FaultTo header addresses:

```
W3CEndpointReference replyToAddress =
"http://myserver.com:7001/myReliableService/myClientCallback";
W3CEndpointReference faultToAddress = "http://myserver.com:7001/myReliableService/FaultTo";
AsyncClientTransportFeature asyncFeature = new AsyncClientTransportFeature(replyToAddress,
faultToAddress);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

The following tables summarizes the constructors that can be used to configure the endpoint reference address for the outgoing ReplyTo and FaultTo headers.

Table 4–6 Constructors for Configuring the ReplyTo and FaultTo Headers

Constructor	Description
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference` *`replyTo`*`)`	Configures the endpoint reference address for the outgoing ReplyTo headers.
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference` *`replyTo`*`, boolean` *`doPublish`*`)`	Configures the following: ■ Endpoint reference address for the outgoing ReplyTo headers. ■ Whether to publish the endpoint at the specified address. For more information, see Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint".
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference` *`replyTo`*`, boolean` *`doPublish`*`, boolean` *`useAsyncWithSyncInvoke`*`)`	Configures the following: ■ Endpoint reference address for the outgoing ReplyTo headers. ■ Whether to publish the endpoint at the specified address. For more information, see Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint". ■ Whether to enable asynchronous client transport for synchronous operations. For more information, see Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations".
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference` *`replyTo`*`, javax.xml.ws.wsaddressing.W3CEndpointReference` *`faultTo`*`)`	Configures the endpoint reference address for the outgoing ReplyTo and FaultTo headers

Table 4–6 (Cont.) Constructors for Configuring the ReplyTo and FaultTo Headers

Constructor	Description
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference` *`replyTo`*`, javax.xml.ws.wsaddressing.W3CEndpointReference` *`faultTo`*`, boolean` *`doPublish`*`)`	Configures the following: ▪ Endpoint reference address for the outgoing ReplyTo and FaultTo headers. ▪ Whether to publish the endpoint at the specified address. For more information, see Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint".
`AsyncClientTransportFeature(javax.xml.ws.wsaddressing.W3CEndpointReference` *`replyTo`*`, javax.xml.ws.wsaddressing.W3CEndpointReference` *`faultTo`*`, boolean` *`doPublish`*`, boolean` *`useAsyncWithSyncInvoke`*`)`	Configures the following: ▪ Endpoint reference address for the outgoing ReplyTo and FaultTo headers. ▪ Whether to publish the endpoint at the specified address. For more information, see Section 4.5.1.4, "Publishing the Asynchronous Response Endpoint". ▪ Whether to enable asynchronous client transport for synchronous operations. For more information, see Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations".

4.5.1.3 Configuring the Context Path of the Asynchronous Response Endpoint

When a client is running within a servlet or Web application-based Web service, it can use its ServletContext and context path to construct the asynchronous response endpoint. You pass the information as an argument to the `AsyncClientTransportFeature`, as follows:

- When running inside a servlet:

```
AsyncClientTransportFeature asyncFeature =
      new AsyncClientTransportFeature(getServletContext());
```

- When running inside a Web service or an EJB-based Web service:

```
import com.sun.xml.ws.api.server.Container;
...
Container c = ContainerResolver.getInstance().getContainer();
ServletContext servletContext = c.getSPI(ServletContext.class);
AsyncClientTransportFeature asyncFeature =
    new AsyncClientTransportFeature(servletContext);
```

The specified context must be scoped within the current application or refer to an unused context; it cannot refer to a context that is scoped to another deployed application.

> **Note:** When you use the empty constructor for `AsyncClientTransportFeature`, the Web services runtime attempts to discover the container in which the current feature was instantiated and publish the endpoint using any available container context.

The following tables summarizes the constructors that can be used to configure the context path of the asynchronous response endpoint.

Table 4–7 Constructors for Configuring the Context Path of the Asynchronous Response Endpoint

Constructor	Description
`AsyncClientTransportFeature(java.lang.Object context)`	Configures the context path of the asynchronous response endpoint.
`AsyncClientTransportFeature(java.lang.Object context, boolean useAsyncWithSyncInvoke)`	Configures the following: ■ Context path of the asynchronous response endpoint. ■ Whether to enable asynchronous client transport for synchronous operations. For more information, see Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations".

4.5.1.4 Publishing the Asynchronous Response Endpoint

You can configure whether to publish the asynchronous response endpoint by passing the `doPublish` boolean value as an argument to `AsycnClientTransportFeature()` when configuring the following properties:

- Address of the asynchronous response endpoint. See Table 4–5.
- ReplyTo and FaultTo headers. See Table 4–6.
- Context path of the asynchronous response endpoint. See Table 4–7.

If `doPublish` is set to false, then the asynchronous response endpoint is not published automatically, but WS-Addressing headers will be added to outbound non-one-way messages. This scenario supports the following programming models:

- Asynchronous polling (with no attempt to access the Response object)
- Dispatch asynchronous polling (with no attempt to access the Response object)
- Dispatch one-way invocation
- Synchronous invocation using synchronous transport option (default)

For all other asynchronous programming models, the availability of a asynchronous response endpoint is required and the Web service client is responsible for publishing it prior to making outbound requests if `doPublish` is set to false.

The following example configures the asynchronous response endpoint address and publishes the asynchronous response endpoint:

```
String responseAddress = "http://localhost:7001/myReliableService/myReliableResponseEndpoint";
boolean doPublish = true;
AsyncClientTransportFeature asyncFeature =
      new AsyncClientTransportFeature(responseAddress, doPublish);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

4.5.1.5 Configuring Asynchronous Client Transport for Synchronous Operations

You can enable or disable asynchronous client transport for synchronous operations using the `useAsyncWithSyncInvoke` boolean flag when configuring the following properties:

- Address of the asynchronous response endpoint. See Table 4–5.
- ReplyTo and FaultTo headers. See Table 4–6.
- Context path of the asynchronous response endpoint. See Table 4–7.

The following example configures the asynchronous response endpoint address and enables use of asynchronous client transport for synchronous operations:

```
String responseAddress = "http://localhost:7001/myReliableService/myReliableResponseEndpoint";
boolean useAsyncWithSyncInvoke = true;
AsyncClientTransportFeature asyncFeature =
      new AsyncClientTransportFeature(responseAddress, useAsyncWithSyncInvoke);
BackendService port = _service.getBackendServicePort(asyncFeature);
```

4.5.2 Developing the Asynchronous Handler Interface

Note: If you set a single asynchronous handler instance on the port, as described in this section, and subsequently attempt to configure a per-request asynchronous handler, as described in Section 4.7, "Using the JAX-WS Reference Implementation", then a runtime exception is returned.

As described in Section 4.4, "Building the Client Artifacts for Asynchronous Web Service Invocation", the asynchronous handler interface, `weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature`, sets a single asynchronous handler instance on the port rather than on a per-request basis.

For example, when you build the client classes using clientgen, as described in Section 4.4, "Building the Client Artifacts for Asynchronous Web Service Invocation", the asynchronous handler interface is generated, as shown below.

Example 4–1 Example of the Asynchronous Handler Interface

```
import javax.xml.ws.Response;


/**
 * This class was generated by the JAX-WS RI.
 * Oracle JAX-WS 2.1.5
 * Generated source version: 2.1
 *
 */
public interface BackendServiceAsyncHandler {


    /**
     *
     * @param response
     */
    public void onDoSomethingResponse(Response<DoSomethingResponse> response);

}
```

The asynchronous handler interface is generated as part of the same package as the port interface and represents the methods required to accept responses for any operation defined on the service. You can import and implement this interface in your client code to provide a way to receive and process asynchronous responses in a strongly-typed manner.

To set a single asynchronous handler instance on the port, pass an instance of the `weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature` as a

parameter when creating the Web service proxy or dispatch. You specify the name of the asynchronous handler that will be invoked when a response message is received.

The following example shows how to develop an asynchronous handler interface. The example demonstrates how to initialize the `AsyncClientHandlerFeature` to connect the asynchronous handler implementation to the port used to make invocations on the backend service. This example is excerpted from Example 3–2, "Asynchronous Web Service Client Best Practices Example".

Example 4–2 Example of Developing the Asynchronous Handler Interface

```
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
...
    BackendServiceAsyncHandler handler = new BackendServiceAsyncHandler() {
        public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
            // ... Handle Response ...
            try {
                DoSomethingResponse response = res.get();
                _lastResponse = response.getReturn();
                System.out.println("Got async response: " + _lastResponse);
            } catch (Exception e) {
                _lastResponse = e.toString();
                e.printStackTrace();
              }
        }
      };
    AsyncClientHandlerFeature handlerFeature = new AsyncClientHandlerFeature(handler);
    features.add(handlerFeature);
    _features = features.toArray(new WebServiceFeature[features.size()]);
    BackendService anotherPort = _service.getBackendServicePort(_features);
...
    // Make the invocation. Our asynchronous handler implementation (set
    // into the AsyncClientHandlerFeature above) receives the response.
    String request = "Dance and sing";
    System.out.println("Invoking DoSomething asynchronously with request: " + request);
    anotherPort.doSomethingAsync(request);
```

4.5.3 Propagating User-defined Request Context to the Response

The `weblogic.wsee.jaxws.JAXWSProperties` API defines the following properties that enables users to propagate user-defined request context information to the response message, without relying on the asynchronous handler instance state.

The asynchronous handler instance may be created at any time; for example, if the client's server goes down and is restarted. Therefore, storing request context in the asynchronous handler interface will not be useful.

The `JAXWSProperties` properties are defined in the following table.

Table 4–8 Properties Supported by the JAXWSProperties API

This property . . .	Specifies . . .
`MESSAGE_ID`	Message ID for the request. The client can set this property on the request context to override the auto-generation of the per-request Message ID header.

Table 4–8 (Cont.) Properties Supported by the JAXWSProperties API

This property . . .	Specifies . . .
PERSISTENT_CONTEXT	Context properties required by the client or the communication channels. Web service clients can persist context properties, as long as they are Serializable, for the request message. These properties will be available in the response context map available from the Response object when the asynchronous handler is invoked. For more information, see Section 4.8, "Propagating Request Context to the Response".
RELATES_TO	Message ID to which the response correlates.
REQUEST_TIMEOUT	For synchronous operations using asynchronous client transport, maximum amount of time to block and wait for a response. This property default to 0 indicating no timeout.

In addition, Web service clients can persist context properties, as long as they are Serializable, for the request message. Context properties can include those required by the client or the communication channels. Message properties can be stored as part of the weblogic.wsee.jaxws.JAXWSProperties.PERSISTENT_CONTEXT Map property and retrieved after the response message is returned. For complete details, see Section 4.8, "Propagating Request Context to the Response".

4.6 Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)

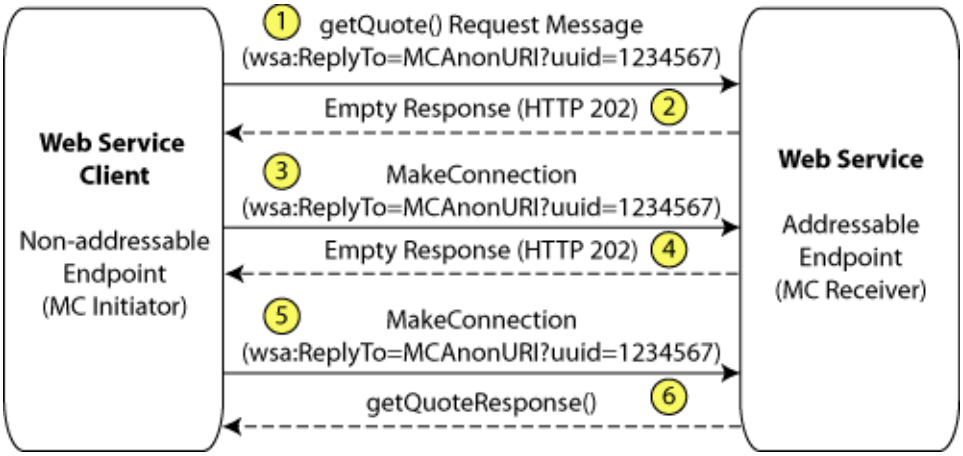
Web Services Make Connection is a client polling mechanism that provides an alternative to asynchronous client transport, typically to provide support for clients that are behind a firewall. WebLogic Server supports WS-MakeConnection version 1.1, as described in the Make Connection specification at: http://docs.oasis-open.org/ws-rx/wsmc/200702, and is backwards compatible with version 1.0.

Specifically, Make Connection:

- Enables the decoupling of the response message from the initiating transport request used to send the request message (similar to asynchronous client transport).
- Supports Web service clients that are non-addressable and unable to accept an incoming connection (for example, clients behind a firewall).
- Enables a Web service client to act as an MC-Initiator and the Web service to act as an MC-Receiver, as defined by the WS-MakeConnection specification.

The following figure, borrowed from the Web Services Make Connection specification, shows a typical Make Connection message flow.

Figure 4–2 Make Connection Message Flow



As shown in the previous figure, the Make Connection message flow is as follows:

1. The `getQuote()` request message is sent from the Web service client (MC Initiator) to the Web service (MC Receiver). The ReplyTo header contains a Make Connection anonymous URI that specifies the UUID for the MC Initiator.

   The MC Receiver receives the `getQuote()` message. The presence of the Make Connection anonymous URI in the ReplyTo header indicates that the response message can be sent back on the connection's back channel or the client will use Make Connection polling to retrieve it.

2. The MC Receiver closes the connection by sending back an empty response (HTTP 202) to the MC Initiator.

   Upon receiving an empty response, the MC Initiator initializes and starts its polling mechanism to enable subsequent polls to be sent to the MC Receiver. Specifically, the MC Initiator polling mechanism starts a timer with expiration set to the interval configured for the time between successive polls.

3. Upon timer expiration, the MC Initiator sends a Make Connection message to the MC Receiver with the same Make Connection anonymous URI information in its message.

4. As the MC Receiver has not completed process the getQuote() operation, no response is available to send back to the MC Initiator. As a result, the MC Receiver closes the connection by sending back another empty response (HTTP 202) indicating that no responses are available at this time.

   Upon receipt of the empty message, the MC Initiator restarts the timer for the Make Connection polling mechanism.

   Before the timer expires, the `getQuote()` operation completes. Since the original request contained a Make Connection anonymous URI in its ReplyTo header, the MC Receiver stores the response and waits to receive the next Make Connection message with a matching address.

5. Upon timer expiration, the MC Initiator sends a Make Connection message to the MC Receiver with the same Make Connection anonymous URI information in its message.

6. Upon receipt of the Make Connection message, the MC Receiver retrieves the stored response message and sends it as a response to the received Make Connection message.

The MC Initiator receives the response message and terminates the Make Connection polling mechanism.

Make Connection transport is recommended when using asynchronous invocation from behind a firewall. For a list of programming models supported, see Table 4–2, " Transport Types for Invoking Web Services Asynchronously".

The following sections describe how to enable and configure Make Connection on a Web service and client:

- Section 4.6.1, "Enabling and Configuring Make Connection on a Web Service"
- Section 4.6.2, "Enabling and Configuring Make Connection on a Web Service Client"

4.6.1 Enabling and Configuring Make Connection on a Web Service

Make Connection can be enabled by attaching a Make Connection policy assertion to the Web service and then calling its methods from a client using the standard JAX-WS client APIs. A policy can be attached to a Web service in one of the following ways:

- Adding an `@Policy` annotation to the JWS file. You can attach a Make Connection policy at the class level only.
- Adding reference to the policy to the Web service WSDL.

The following sections describe the steps required to enable Make Connection on a Web service:

- Section 4.6.1.1, "Creating the Web Service Make Connection WS-Policy File (Optional)"
- Section 4.6.1.2, "Programming the JWS File to Enable Make Connection"

4.6.1.1 Creating the Web Service Make Connection WS-Policy File (Optional)

A WS-Policy file is an XML file that contains policy assertions that comply with the WS-Policy specification. In this case, the WS-Policy file contains Web service Make Connection policy assertions.

WebLogic Server includes pre-packaged WS-Policy files that contain typical Make Connection assertions that you can use if you do not want to create your own WS-Policy file. The pre-packaged WS-Policy files that support Make Connection are listed in the following table. In some cases, both reliable messaging and Make Connection are enabled by the policy. For more information, see Appendix A, "Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection".

Note: You can attach Make Connection policies at the class level only; you cannot attach the Make Connection policies at the method level.

Table 4–9 Pre-packaged WS-Policy Files That Support Make Connection

Pre-packaged WS-Policy File	Description
`Mc1.1.xml`	Enables Make Connection support on the Web service and specifies usage as optional on the Web service client. The WS-Policy 1.5 protocol is used. See Section A.5, "Mc1.1.xml (WS-Policy File)".

Table 4–9 (Cont.) Pre-packaged WS-Policy Files That Support Make Connection

Pre-packaged WS-Policy File	Description
`Mc.xml`	Enables Make Connection support on the Web service and specifies usage as optional on the Web service client. The WS-Policy 1.2 protocol is used. See Section A.6, "Mc.xml (WS-Policy File)".
`Reliability1.2_ExactlyOnce_WithMC1.1.xml`	Specifies policy assertions related to quality of service. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client. See Section A.7, "Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File)".
`Reliability1.2_SequenceSTR.xml`	Specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client. See Section A.8, "Reliability1.2_SequenceSTR.xml (WS-Policy File)".
`Reliability1.0_1.2.xml`	Combines 1.2 and 1.0 WS-Reliable Messaging policy assertions. The policy assertions for the 1.2 version Make Connection support on the Web service and specifies usage as optional on the Web service client. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. See Section A.12, "Reliability1.0_1.2.xml (WS-Policy File)".

You can use one of the pre-packaged Make Connection WS-Policy files included in WebLogic Server; these files are adequate for most use cases. You cannot modify the pre-packaged files. If the values do not suit your needs, you must create a custom WS-Policy file. For example, you may wish to configure support of Make Connection as required on the Web service client side. The Make Connection policy assertions conform to the WS-PolicyAssertions specification at `https://www.ibm.com/developerworks/library/specification/ws-polas`.

To create a custom WS-Policy file that contains Make Connection assertions, use the following guidelines:

- The root element of a WS-Policy file is always `<wsp:Policy>`.
- To configure Web service Make Connection, you simply add a `<wsmc:MCSupported>` child element to define the Web service Make Connection support.
- The `<wsmc:MCSupported>` child element contains one policy attribute, `Optional`, that specifies whether Make Connection must be configured on the Web service client. This attribute can be set to `true` or `false`, and is set to `true` by default. If set to `false`, then use of Make Connection is required and both the ReplyTo and FaultTo (if specified) headers must contain Make Connection anonymous URIs.

The following example enables Make Connection on the Web service and specifies that Make Connection must be enabled on the Web service client. In this example, the WS-Policy 1.5 protocol is used.

```
<?xml version="1.0"?>
<wsp15:Policy  xmlns:wsp15="http://www.w3.org/ns/ws-policy"
  xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702">
  <wsmc:MCSupported wsp15:Optional="false" />
</wsp15:Policy>
```

4.6.1.2 Programming the JWS File to Enable Make Connection

This section describes how to enable Make Connection on the Web service using a pre-packaged or custom Make Connection WS-Policy file. For information about creating a custom policy file, see Section 4.6.1.1, "Creating the Web Service Make Connection WS-Policy File (Optional)".

Use the `@Policy` annotation in your JWS file to specify that the Web service has a WS-Policy file attached to it that contains Make Connection assertions. WebLogic Server delivers a set of pre-packaged WS-Policy files, as described in Appendix A, "Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection".

Refer to the following guidelines when using the `@Policy` annotation for Web service reliable messaging:

- You can attach the Make Connection policy at the class level only; you cannot attach the Make Connection policy at the method level.
- Use the `uri` attribute to specify the build-time location of the policy file, as follows:
  - If you have created your own WS-Policy file, specify its location relative to the JWS file. For example:

    ```
    @Policy(uri="McPolicy.xml", attachToWsdl=true)
    ```

    In this example, the `McPolicy.xml` file is located in the same directory as the JWS file.
  - To specify one of the pre-packaged WS-Policy files or a WS-Policy file that is packaged in a shared Java EE library, use the `policy:` prefix along with the name and path of the policy file. This syntax tells the `jwsc` Ant task at build-time *not* to look for an actual file on the file system, but rather, that the Web service will retrieve the WS-Policy file from WebLogic Server at the time the service is deployed.

    Note: Shared Java EE libraries are useful when you want to share a WS-Policy file with multiple Web services that are packaged in different Enterprise applications. As long as the WS-Policy file is located in the `META-INF/policies` or `WEB-INF/policies` directory of the shared Java EE library, you can specify the policy file in the same way as if it were packaged in the same archive at the Web service. See "Creating Shared Java EE Libraries and Optional Packages" in *Developing Applications for Oracle WebLogic Server* for information about creating libraries and setting up your environment so the Web service can locate the policy files.
  - To specify that the policy file is published on the Web, use the `http:` prefix along with the URL, as shown in the following example:

    ```
    @Policy(uri="http://someSite.com/policies/mypolicy.xml"
            attachToWsdl=true)
    ```
- Set the `attachToWsdl` attribute of the `@Policy` annotation to specify whether the policy file should be attached to the WSDL file that describes the public contract of the Web service. Typically, you want to publicly publish the policy so that client applications know the reliable messaging capabilities of the Web service. For this reason, the default value of this attribute is `true`.

For more information about the `@Policy` annotation, see "weblogic.jws.Policy" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

The following example shows a simple JWS file that enables Make Connection; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.async

import javax.jws.WebMethod;
import javax.jws.WebService;
import weblogic.jws.Policy;

/**
 * Simple reliable Web Service.
 */

@WebService(name="HelloWorldPortType",
              serviceName="HelloWorldService")

@Policy(uri="McPolicy.xml", attachToWsdl=true)
public class HelloWorldImpl {
  private static String onewaySavedInput = null;

/**
 * A one-way helloWorld method that saves the given string for later
 * concatenation to the end of the message passed into helloWorldReturn.
 */
  @WebMethod()
  public void helloWorld(String input) {
    System.out.println(" Hello World " + input);
    onewaySavedInput = input;
   }

/**
 * This echo method concatenates the saved message from helloWorld
 * onto the end of the provided message, and returns it.
 */
  @WebMethod()
  public String echo(String input2) {
    System.out.println(" Hello World " + input2 + onewaySavedInput);
    return input + onewaySavedInput;
  }
}
```

As shown in the previous example, the custom `McPolicy.xml` policy file is attached to the Web service at the class level, which means that the policy file is applied to all public operations of the Web service. You can attach a Make Connection policy at the class level only; you cannot attach a Make Connection policy at the method level.

The policy file is attached to the WSDL file. For information about the pre-packaged policies available and creating a custom policy, see Section 4.6.1.1, "Creating the Web Service Make Connection WS-Policy File (Optional)".

The `echo()` method has been marked with the `@WebMethod` JWS annotation, which means it is a public operation called `echo`. Because of the `@Policy` annotation, the operation using Make Connection transport protocol.

4.6.2 Enabling and Configuring Make Connection on a Web Service Client

> **Note:** The Make Connection and asynchronous client transport features are mutually exclusive. If you attempt to enable both features on the same Web service client, an error is returned. For more information about asynchronous client transport, see Section 4.5, "Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)".
>
> It is recommended that you use the asynchronous handler feature, `AsyncClientHandlerFeature` when using the asynchronous callback handler programming model. For more information, see Section 4.5.2, "Developing the Asynchronous Handler Interface".

To enable Make Connection on a Web service client, pass an instance of the `weblogic.wsee.mc.api.McFeature` as a parameter when creating the Web service proxy or dispatch. A simple example of how to enable Make Connection is shown below.

> **Note:** This example will use synchronous transport for synchronous methods. To configure Make Connection as the transport for synchronous methods, see Section 4.6.2.4, "Configuring Make Connection as the Transport for Synchronous Methods."

```
package examples.webservices.myservice.client;

import weblogic.wsee.mc.api.McFeature;
...
    List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();
...
    McFeature mcFeature = new McFeature();
    features.add(mcFeature);
...
    // ... Implement asynchronous handler interface as described in
    // Section 4.5.2, "Developing the Asynchronous Handler Interface."
    // ....
    AsyncClientHandlerFeature handlerFeature = new AsyncClientHandlerFeature(handler);
    features.add(handlerFeature);
    _features = features.toArray(new WebServiceFeature[features.size()]);
    BackendService port = _service.getBackendServicePort(_features);
...
    // Make the invocation. Our asynchronous handler implementation (set
    // into the AsyncClientHandlerFeature above) receives the response.
    String request = "Dance and sing";
    System.out.println("Invoking DoSomething asynchronously with request: " + request);
    anotherPort.doSomethingAsync(request);
..
  }
}
```

To configure specific features of Make Connection on the Web service client, as described in the following sections.

- Section 4.6.2.1, "Configuring the Expiration Time for Sending Make Connection Messages"

- Section 4.6.2.2, "Configuring the Polling Interval"
- Section 4.6.2.3, "Configuring the Exponential Backoff"
- Section 4.6.2.4, "Configuring Make Connection as the Transport for Synchronous Methods"

4.6.2.1 Configuring the Expiration Time for Sending Make Connection Messages

Table 4–10 defines that `McFeature` methods for configuring the maximum interval of time before an MC Initiator stops sending Make Connection messages to an MC Receiver.

Table 4–10 Methods for Configuring the Expiration Time for Sending Make Connection Messages

Method	Description
`String getsExpires()`	Returns the expiration value currently configured.
`void setExpires(String` *`expires`*`)`	Set the expiration time. The value specified must be a positive value and conform to the XML schema duration lexical format, `P`*`n`*`Y`*`n`*`M`*`n`*`DT`*`n`*`H`*`n`*`M`*`n`*`S`, where *`n`*`Y` specifies the number of years, *`n`*`M` specifies the number of months, *`n`*`D` specifies the number of days, `T` is the date/time separator, *`n`*`H` specifies the number of hours, *`n`*`M` specifies the number of minutes, and *`n`*`S` specifies the number of seconds. This value defaults to `P1D` (1 day).

4.6.2.2 Configuring the Polling Interval

Table 4–11 defines that `McFeature` methods for configuring the interval of time that must pass before a Make Connection message is sent by an MC Initiator to an MC Receiver after the receipt of an empty response message. If the MC Initiator does not receive a non-empty response for a given message within the specified interval, the MC Initiator sends another Make Connection message.

Table 4–11 Methods for Configuring the Polling Interval

Method	Description
`String getInterval()`	Gets the polling interval.
`void setInterval(String` *`pollingInterval`*`)`	Set the polling interval. The value specified must be a positive value and conform to the XML schema duration lexical format, `P`*`n`*`Y`*`n`*`M`*`n`*`DT`*`n`*`H`*`n`*`M`*`n`*`S`, where *`n`*`Y` specifies the number of years, *`n`*`M` specifies the number of months, *`n`*`D` specifies the number of days, `T` is the date/time separator, *`n`*`H` specifies the number of hours, *`n`*`M` specifies the number of minutes, and *`n`*`S` specifies the number of seconds. This value defaults to `P0DT5S` (5 seconds).

In the following example, the polling interval is set to 36 hours.

```
...
    McFeature mcFeature = new McFeature();
    mcFeature.setInterval("P0DT36H")
    MyService port = service.getMyServicePort(mcFeature);
```

```
...
```

4.6.2.3 Configuring the Exponential Backoff

Table 4–12 defines the `McFeature` methods for configuring the exponential backoff flag. This flag specifies whether the polling interval, described in Section 4.6.2.2, "Configuring the Polling Interval", will be adjusted using the exponential backoff algorithm. In this case, if the MC Initiator does not receive a non-empty response for the time interval specified by the polling interval, the exponential backoff algorithm is used for timing successive retransmissions by the MC Initiator, should the response not be received.

The exponential backoff algorithm specifies that successive polling intervals should increase exponentially, based on the polling interval. For example, if the polling interval is 2 seconds, and the exponential backoff element is set, successive polling intervals if the response is not received are 2, 4, 8, 16, 32, and so on.

This value defaults to false, the same polling interval is used in successive retries; the interval does not increase exponentially.

Table 4–12 Methods for Configuring the Exponential Backoff

Method	Description
`boolean isExponentialBackoff()`	Returns a boolean value indicating whether exponential backoff is enabled.
`void setExponentialBackoff(boolean` *`backoff`*`)`	Set the exponential backoff flag. Valid values are `true` and `false`. This flag defaults to `false`.

In the following example, enables the exponential backoff flag.

```
...
    McFeature mcFeature = new McFeature();
    mcFeature.setMessageInterval(P0DT36H)
    mcFeature.setExponentialBackoff(true);
    MyService port = service.getMyServicePort(mcFeature);
...
```

4.6.2.4 Configuring Make Connection as the Transport for Synchronous Methods

By default, synchronous methods use synchronous transport even when Make Connection is enabled on the client. You can configure your client to use Make Connection as the transport for synchronous methods. In this case, Make Connection messages are sent by the MC Initiator based on the configured polling interval (described in Section 4.6.2.2, "Configuring the Polling Interval") until a non-empty response message is received.

To configure Make Connection as the transport protocol to use for synchronous methods, use one of the following methods:

- When instantiating a new `McFeature()` object, you can pass as a parameter a boolean value that specifies whether Make Connection should be used as the transport protocol for synchronous methods. For example:

```
...
    McFeature mcFeature = new McFeature(true);
    MyService port = service.getMyServicePort(mcFeature);
...
```

- Use the `McFeature` methods defined in Table 4–13. For example:

```
...
    McFeature mcFeature = new McFeature();
    mcFeature.setUseMCWithSyncInvoke(true);
    MyService port = service.getMyServicePort(mcFeature);
...
```

Table 4–13 Methods for Configuring Synchronous Method Support

Method	Description
`boolean isUseMCWithSyncInvoke()`	Returns a boolean value indicating whether synchronous method support is enabled.
`void setUseMCWithSyncInvoke(boolean` *`useMCWithSyncInvoke`*`)`	Sets the synchronous method support flag. Valid values are `true` and `false`. This flag defaults to `false`.

You can set the maximum amount of time a synchronous method will block and wait for a response using the `weblogic.wsee.jaxws.JAXWSProperties.REQUEST_TIMEOUT` property. This property default to 0 indicating no timeout. For more information about setting message properties, see Section 4.5.3, "Propagating User-defined Request Context to the Response".

4.7 Using the JAX-WS Reference Implementation

The JAX-WS Reference Implementation (RI) supports the following programming models:

- Asynchronous client polling through use of the `java.util.concurrent.Future` interface.
- Asynchronous callback handlers on a per request basis. The calling client specifies the callback handler at invocation time. When the response is available, the callback handler is invoked to process the response.

Unlike with asynchronous client transport feature, the JAX-WS RI provides very limited support for WS-Addressing, including:

- Manual support for adding client-side outbound WS-Addressing headers.
- Manual support for publishing the client-side endpoint to receive responses.
- No support for detecting incorrect client-side programming model (resulting in synchronous call hanging, for example).
- No support for surviving a client-side or service-side restart.

The following example shows a simple client file, `AsyncClient`, that has a single method, `AddNumbersTestDrive`, that asynchronously invokes the `AddNumbersAsync` method of the `AddNumbersService` service. The Java code in **bold** is described following the code sample.

```
package examples.webservices.async.client;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;

import javax.xml.ws.BindingProvider;
```

```
import java.util.concurrent.Future;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

public class AsyncClient  {

   private AddNumbersPortType port = null;
   protected void setUp() throws Exception {
      AddNumbersService service = new AddNumbersService();
      port = service.getAddNumbersPort();
      String serverURI = System.getProperty("wls-server");
      ((BindingProvider) port).getRequestContext().put(
            BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
             "http://" + serverURI + "/JAXWS_ASYNC/AddNumbersService");
   }

/**
*
* Asynchronous callback handler
*/
   class AddNumbersCallbackHandler implements AsyncHandler<AddNumbersResponse> {
      private AddNumbersResponse output;
      public void handleResponse(Response<AddNumbersResponse> response) {
         try {
            output = response.get();
         } catch (ExecutionException e) {
             e.printStackTrace();
         } catch (InterruptedException e) {
              e.printStackTrace();
          }
      }
      AddNumbersResponse getResponse() {
         return output;
      }
   }

   public void AddNumbersTestDrive() throws Exception {
      int number1 = 10;
      int number2 = 20;

      // Asynchronous Callback method
      AddNumbersCallbackHandler callbackHandler =
         new AddNumbersCallbackHandler();
      Future<?> resp = port.addNumbersAsync(number1, number2,
         callbackHandler);
      // For the purposes of a test, block until the async call completes
      resp.get(5L, TimeUnit.MINUTES);
      int result = callbackHandler.getResponse().getReturn();

      // Polling method
      Response<AddNumbersResponse> addNumbersResp =
           port.AddNumbersAsync(number1, number2);
      while (!addNumbersResp.isDone()) {
            Thread.sleep(100);
      }
      AddNumbersResponse reply = addNumbersResp.get();
      System.out.println("Server responded through polling with: " +
          reply.getResponseType());
   }
}
```

The example demonstrates the steps to implement both the asynchronous polling and asynchronous callback handler programming models.

To implement an asynchronous callback handler:

1. Create an asynchronous handler that implements the `javax.xml.ws.AsyncHandler<T>` interface (see http://download.oracle.com/javaee/5/api/javax/xml/ws/AsyncHandler.html). The asynchronous handler defines one method, `handleResponse`, that enables clients to receive callback notifications at the completion of service endpoint operations that are invoked asynchronously. The type should be set to `AddNumberResponse`.

```
class AddNumbersCallbackHandler implements AsyncHandler<AddNumbersResponse> {
   private AddNumbersResponse output;

   public void handleResponse(Response<AddNumbersResponse> response) {
      try {
         output = response.get();
         } catch (ExecutionException e) {
           e.printStackTrace();
         } catch (InterruptedException e) {
           e.printStackTrace();

         }
      }

      AddNumbersResponse getResponse() {
         return output;
   }
}
```

2. Instantiate the asynchronous callback handler.

```
AddNumbersCallbackHandler callbackHandler =
   new AddNumbersCallbackHandler();
```

3. Instantiate the `AddNumbersService` Web service and call the asynchronous version of the Web service method, `addNumbersAsync`, passing a handle to the asynchronous callback handler.

```
AddNumbersService service = new AddNumbersService();
port = service.getAddNumbersPort();
...

Future<?> resp = port.addNumbersAsync(number1, number2,
   callbackHandler);
```

   `java.util.concurrent.Future` (see http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Future.html) represents the result of an asynchronous computation and provides methods for checking the status of the asynchronous task, getting the result, or canceling the task execution.

4. Get the result of the asynchronous computation. In this example, a timeout value is specified to wait for the computation to complete.

```
resp.get(5L, TimeUnit.MINUTES);
```

5. Use the callback handler to access the response message.

```
int result = callbackHandler.getResponse().getReturn();
```

To implement an asynchronous polling mechanism:

1. Instantiate the `AddNumbersService` Web service and call the asynchronous version of the Web service method, `addNumbersAsync`.

   ```
   Response<AddNumbersResponse> addNumbersResp =
        port.AddNumbersAsync(number1, number2);
   ```

2. Sleep until a message is received.

   ```
   while (!addNumbersResp.isDone()) {
           Thread.sleep(100);
   ```

3. Poll for a response.

   ```
   AddNumbersResponse reply = addNumbersResp.get();
   ```

4.8 Propagating Request Context to the Response

WebLogic Server provides a powerful facility that enables you to attach your business context—for example, a business-level message ID—to the request message and access it when the response is returned, regardless of what the request and response messages convey over the wire. For example, you may have a business-level message ID that will not otherwise be available in the response message. By propagating this information with the message, you can access it when the response message is returned.

Web service clients can store any request message context property, as long as it is Serializable. Message context properties can be stored as part of the `weblogic.wsee.jaxws.JAXWSProperties.PERSISTENT_CONTEXT` Map property and retrieved after the response message is returned.

The following example shows how to use the `PERSISTENT_CONTEXT` Map property to define and set a message context property.

Example 4–3 Setting Message Context Properties

```
import weblogic.wsee.jaxws.JAXWSProperties;
. . .
MyClientPort port = myService.getPort();
Map<String, Serializable> clientPersistProps =
    port.getRequestContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
Serializable obj = <my_property>;
clientPersistProps.put("MyProperty", obj);

port.myOperationAsync(<args>, new AsyncHandler<MyOperationResponse>() {
    public void handleResponse(Response<MyOperationResponse> res) {
        try {
            // Get the actual response
            MyOperationResponse response = res.get().getReturn();

            // Get the property stored when making request. Note, this property did not get
            // passed over the wire with the reuqest. The Web services runtime stores it.
            Map<String, Serializable> clientPersistProps =
                res.getContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
            Serializable obj = clientPersistProps.get("MyProperty");
            // Do something with MyProperty
        } catch (Exception e) {
```

```
            // Error handling
        }
    }
});
...
```

4.9 Monitoring Asynchronous Web Service Invocation

You can monitor runtime information for clients that invoke Web services asynchronously, such as number of invocations, errors, faults, and so on, using the Administration Console. To monitor Web service clients, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the Web service client is packaged. Expand the application by clicking the **+** node and click on the application module within which the Web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab.

If you use the Make Connection transport protocol, you can monitor the Make Connection anonymous endpoints for a Web service or client. For each anonymous endpoint, runtime monitoring information is displayed, such as the number of messages received, the number of messages pending, and so on.

You can customize the information that is shown in the table by clicking **Customize this table**.

To monitor Make Connection anonymous endpoints for a Web service, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the Web service is packaged. Expand the application by clicking the **+** node; the Web services in the application are listed under the **Web Services** category. Click on the name of the Web service and select **Monitoring> Ports> Make Connection**.

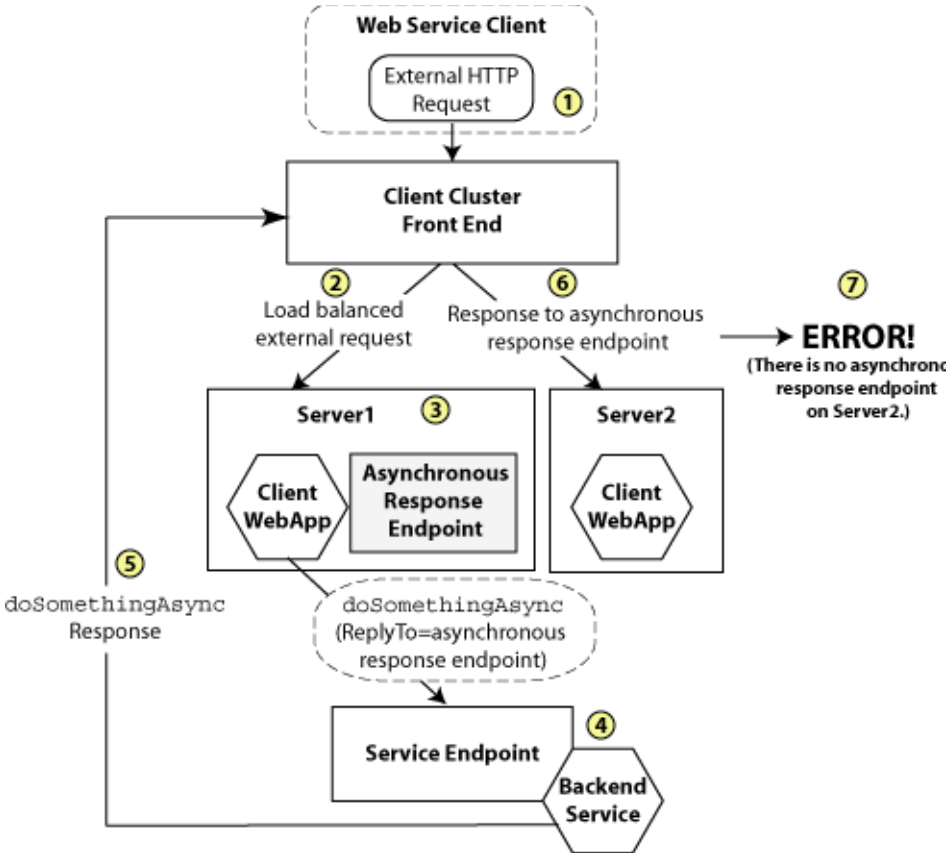
To monitor Make Connection anonymous endpoints for a Web service client, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the Web service client is packaged. Expand the application by clicking the **+** node and click on the application module within which the Web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab. Then click **Monitoring> Servers> Make Connection**.

4.10 Clustering Considerations for Asynchronous Web Service Messaging

When a Web service client runs in a cluster, you need to make special allowances to ensure that the response messages can be delivered properly to the asynchronous response endpoint for asynchronous calls. You defined the asynchronous response endpoint with the AsyncClientTransportFeature, as described in Section 4.5.1, "Enabling and Configuring the Asynchronous Client Transport Feature".

Consider the scenario shown in the following figure.

Figure 4–3 Clustering Scenario Resulting in an Error



In the scenario shown in the previous figure:

- A two-node cluster hosts the client application; the nodes are named Server1 and Server2. The cluster has a simple load-balancing front-end proxy.
- The client application is a Web application called ClientWebApp which is deployed homogeneously to the cluster. In other words, the Web application runs on both member servers in the cluster.
- External clients of the ClientWebApp application make requests through the cluster front-end address.

Now consider the following sequence:

1. An external client requests a page from ClientWebApp via the cluster front-end.
2. The cluster front-end load balances the page request and sends it to the ClientWebApp on Server1.
3. ClientWebApp on Server1 creates an instance of a Web service client, BackendServiceClient, to communicate with its back-end service, BackendService. The creation of BackendServiceClient causes an asynchronous response endpoint to be published to receive asynchronous responses whenever BackendServiceClient is used to make an asynchronous request.
4. ClientWebApp on Server1 calls `BackendServiceClient.doSomethingAsync()` to perform an operation on the backend service. The address of the asynchronous response endpoint is included in the ReplyTo address. This address starts with the address of the cluster front end, and not the address of Server1.

5. The cluster receives the response to the `doSomething` operation.
6. The cluster load balances the message, this time to Server2.
7. The message delivery fails because there is no asynchronous response endpoint on Server2 to receive the response.

You can use one of the following to resolve this problem:

- Use a SOAP-aware cluster front-end proxy plug-in, such as WebLogic Server HttpClusterServlet. For more information, see "Configure Proxy Plug-ins" in *Using Clusters for Oracle WebLogic Server*. This option may not be feasible, for example if your company has standardized on a cluster front-end technology.
- Ensure that all member servers in a cluster publish an asynchronous response endpoint so that the asynchronous response messages can be delivered to any member server and optionally forwarded to the correct server via in-place cluster routing.

To implement the second option, it is recommended that you define a singleton port instance and initialize it when the client container initializes (upon deployment). For an example illustrating the recommended method for initializing the asynchronous response endpoint in a cluster, see Example 3–2, "Asynchronous Web Service Client Best Practices Example".

Note: You may choose to initialize the endpoint in different ways depending on the container type. For example, if the client is hosted in a Web service, a method on the Web service container could be annotated with `@PostConstruct` and that method could initialize the singleton port. In an EJB container, you could use the `ejbCreate()` method as the trigger point for creating the singleton port.

5

Roadmap for Developing Reliable Web Services and Clients

This chapter presents best practices for developing WebLogic Web services and clients for Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

- Section 5.1, "Roadmap for Developing Reliable Web Service Clients"
- Section 5.2, "Roadmap for Developing Reliable Web Services"
- Section 5.3, "Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection)"
- Section 5.4, "Roadmap for Securing Reliable Web Services"

> **Note:** It is assumed that you are familiar with the general concepts for developing Web services and clients, as described in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
>
> See also Section 7.2, "Roadmap for Configuring Web Service Persistence.".

5.1 Roadmap for Developing Reliable Web Service Clients

Table 5–1 provides best practices for developing reliable Web service clients, including an example that illustrates the best practices presented. These guidelines should be used in conjunction with the guidelines provided in Chapter 3, "Roadmaps for Developing Web Service Clients.".

Table 5–1 Roadmap for Developing Reliable Web Service Clients

Best Practice	Description
Always implement a reliability error listener.	For more information, see Section 6.8, "Implementing the Reliability Error Listener."
Group messages into *units of work*.	Rather than incur the RM sequence creation and termination protocol overhead for *every* message sent, you can group messages into business units of work—also referred to as *batching*. For more information, see Section 6.11, "Grouping Messages into Business Units of Work (Batching)". **Note**: This best practice is not demonstrated in Example 5–1.
Set the acknowledgement interval to a realistic value for your particular scenario.	The recommended setting is two times the nominal interval between requests. For more information, see Section 6.7.9, "Configuring the Acknowledgement Interval." **Note**: This best practice is not demonstrated in Example 5–1.
Set the base retransmission interval to a realistic value for your particular scenario.	The recommended setting is two times the acknowledgement interval or nominal response time, whichever is greater. For more information, see Section 6.7.4, "Configuring the Base Retransmission Interval." **Note**: This best practice is not demonstrated in Example 5–1.
Set timeouts (inactivity and sequence expiration) to realistic values for your particular scenario.	For more information, see Section 6.7.7, "Configuring Inactivity Timeout" and Section 6.7.6, "Configuring the Sequence Expiration." **Note**: This best practice is not demonstrated in Example 5–1.

The following example illustrates best practices for developing reliable Web service clients.

Example 5–1 Reliable Web Service Client Best Practices Example

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.xml.bind.JAXBContext;
import javax.xml.ws.*;

import weblogic.jws.jaxws.client.ClientIdentityFeature;
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
import weblogic.jws.jaxws.client.async.AsyncClientTransportFeature;
import weblogic.wsee.reliability2.api.ReliabilityErrorContext;
import weblogic.wsee.reliability2.api.ReliabilityErrorListener;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;

import com.sun.xml.ws.developer.JAXWSProperties;

/**
 * Example client for invoking a reliable Web service asynchronously.
 */
public class BestPracticeAsyncRmClient
  extends GenericServlet {

  private BackendReliableServiceService _service;
  private BackendReliableService _singletonPort;
  private WebServiceFeature[] _features;

  private static int _requestCount;
  private static String _lastResponse;
```

```
private static final String MY_PROPERTY = "MyProperty";

@Override
public void init()
  throws ServletException {

  _requestCount = 0;
  _lastResponse = null;

  // Only create the Web service object once as it is expensive to create repeatedly.
  if (_service == null) {
    _service = new BackendReliableServiceService();
  }

  // Best Practice: Use a stored list of features, per client ID, to create client instances.
  // Define all features for the Web service port, per client ID, so that they are
  // consistent each time the port is called. For example:
  // _service.getBackendServicePort(_features);

  List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

  // Best Practice: Explicitly define the client ID.
  ClientIdentityFeature clientIdFeature =
    new ClientIdentityFeature("MyBackendServiceAsyncRmClient");
  features.add(clientIdFeature);

  // Best Practice: Always implement a reliability error listener.
  // Include this feature in your reusable feature list. This enables you to determine
  // a reason for failure, for example, RM cannot deliver a request or the RM sequence fails in
  // some way (for example, client credentials refused at service).
  WsrmClientInitFeature rmFeature = new WsrmClientInitFeature();
  features.add(rmFeature);
  rmFeature.setErrorListener(new ReliabilityErrorListener() {
    public void onReliabilityError(ReliabilityErrorContext context) {

      // At a *minimum* do this
      System.out.println("RM sequence failure: " +
                         context.getFaultSummaryMessage());
      _lastResponse = context.getFaultSummaryMessage();

      // And optionally do this...

      // The context parameter conveys whether a request or the entire
      // sequence has failed. If a sequence fails, you will get a notification
      // for each undelivered request (if any) on the sequence.
      if (context.isRequestSpecific()) {
        // Single request failure (possibly as part of a larger sequence failure).
        // Retrieve the original request.
        String operationName = context.getOperationName();
        System.out.println("Failed to deliver request for operation '" +
                           operationName + "'. Fault summary: " +
                           context.getFaultSummaryMessage());
        if ("DoSomething".equals(operationName)) {
          try {
            String request = context.getRequest(JAXBContext.newInstance(),
                                                String.class);
            System.out.println("Failed to deliver request for operation '" +
                               operationName + "' with content: " +
                               request);
            Map<String, Serializable> requestProps =
```

```
            context.getUserRequestContextProperties();
          if (requestProps != null) {
            // Retrieve the request property. Use MyProperty
            // to describe the request that failed and print this value
            // during the simple 'error recovery' below.
            String myProperty = (String)requestProps.get(MY_PROPERTY);
            System.out.println("Got MyProperty value propagated from request: "+
                               myProperty);
            System.out.println(myProperty + " failed!");
          }
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    } else {
      // The entire sequence has encountered an error.
      System.out.println("Entire sequence failed: " +
                         context.getFaultSummaryMessage());

    }
  }
});

// Asynchronous endpoint.
AsyncClientTransportFeature asyncFeature =
  new AsyncClientTransportFeature(getServletContext());
features.add(asyncFeature);

// Best Practice: Define a port-based asynchronous callback handler,
// AsyncClientHandlerFeature, for asynchronous and dispatch callback handling.
BackendReliableServiceAsyncHandler handler =
  new BackendReliableServiceAsyncHandler() {
    public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
      // ... Handle Response ...
      try {
        // Show getting the MyProperty value back.
        DoSomethingResponse response = res.get();
        _lastResponse = response.getReturn();
        System.out.println("Got (reliable) async response: " + _lastResponse);
        // Retrieve the request property. This property can be used to
        // 'remember' the context of the request and subsequently process
        // the response.
        Map<String, Serializable> requestProps =
          (Map<String, Serializable>)
            res.getContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
        String myProperty = (String)requestProps.get(MY_PROPERTY);
        System.out.println("Got MyProperty value propagated from request: "+
                           myProperty);
      } catch (Exception e) {
        _lastResponse = e.toString();
        e.printStackTrace();
      }
    }
  };
AsyncClientHandlerFeature handlerFeature =
  new AsyncClientHandlerFeature(handler);
features.add(handlerFeature);

// Set the features used when creating clients with
// the client ID "MyBackendServiceAsyncRmClient."
```

```
  _features = features.toArray(new WebServiceFeature[features.size()]);

  // Best Practice: Define a singleton port instance and initialize it when
  // the client container initializes (upon deployment).
  // The singleton port will be available for the life of the servlet.
  // Creation of the singleton port triggers the asynchronous response endpoint to be published
  // and it will remain published until our container (Web application) is undeployed.
  // Note, we will get a call to destroy() before this.
  _singletonPort = _service.getBackendReliableServicePort(_features);
}

@Override
public void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException {

  // TODO: ... Read the servlet request ...

  // For this simple example, echo the _lastResponse captured from
  // an asynchronous DoSomethingResponse response message.

  if (_lastResponse != null) {
    res.getWriter().write(_lastResponse);
    _lastResponse = null; // Clear the response so we can get another
    return;
  }

  // Set _lastResponse to NULL in order to make a new invocation against
  // BackendService to generate a new response

  // Best Practice: Synchronize use of client instances.
  // Create another port using the *exact* same features used when creating _singletonPort.
  // Note, this port uses the same client ID as the singleton port and it is effectively the
  // same as the singleton from the perspective of the Web services runtime.
  // This port will use the asynchronous response endpoint for the client ID,
  // as it is defined in the _features list.
  // NOTE: This is *DEFINITELY* not best practice or ideal because our application is
  //       incurring the cost of an RM handshake and sequence termination
  //       for *every* reliable request sent. It would be better to send
  //       multiple requests on each sequence. If there is not a natural grouping
  //       for messages (a business 'unit of work'), then you could batch
  //       requests onto a sequence for efficiency. For more information, see
  //       Section 6.11, "Grouping Messages into Business Units of Work (Batching)."
  BackendReliableService anotherPort =
    _service.getBackendReliableServicePort(_features);

  // Set the endpoint address for BackendService.
  ((BindingProvider)anotherPort).getRequestContext().
    put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        "http://localhost:7001/BestPracticeReliableService/BackendReliableService");

  // Make the invocation. Our asynchronous handler implementation (set
  // into the AsyncClientHandlerFeature above) receives the response.
  String request = "Protect and serve";
  System.out.println("Invoking DoSomething reliably/async with request: " +
                     request);
  // Add a persistent context property that will be returned on the response.
  // This property can be used to 'remember' the context of this
  // request and subsequently process the response. This property will *not*
  // get passed over wire, so the properties can change independent of the
```

```
    // application message.
    Map<String, Serializable> persistentContext =
      (Map<String, Serializable>)((BindingProvider)anotherPort).
        getRequestContext().get(JAXWSProperties.PERSISTENT_CONTEXT);
    String myProperty = "Request " + (++_requestCount);
    persistentContext.put(MY_PROPERTY, myProperty);
    System.out.println("Request being made (reliably) with MyProperty value: " +
                       myProperty);
    anotherPort.doSomethingAsync(request);

    // Return a canned string indicating the response was not received
    // synchronously. Client needs to invoke the servlet again to get
    // the response.
    res.getWriter().write("Waiting for response...");

    // Best Practice: Explicitly close client instances when processing is complete.
    // If not closed, the port will be closed automatically when it goes out of scope.
    // This will force the termination of the RM sequence we created when sending the first
    // doSomething request. For a better way to handle this, see
    // Section 6.11, "Grouping Messages into Business Units of Work (Batching)."
    // NOTE: Even though the port is closed explicitly (or even if it goes out of scope)
    //       the reliable request sent above will still be delivered
    //       under the scope of the client ID used. So, even if the service endpoint
    //       is down, RM retries the request and delivers it when the service endpoint
    //       available. The asynchronous resopnse will be delivered as if the port instance was
    //       still available.
    ((java.io.Closeable)anotherPort).close();
  }

  @Override
  public void destroy() {

    try {
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the singleton port created during initialization. Note, the asynchronous
      // response endpoint generated by creating _singletonPort *remains*
      // published until our container (Web application) is undeployed.
      ((java.io.Closeable)_singletonPort).close();
      // Upon return, the Web application is undeployed, and our asynchronous
      // response endpoint is stopped (unpublished). At this point,
      // the client ID used for _singletonPort will be unregistered and will no longer be
      // visible from the Administration Console and WLST.
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

5.2 Roadmap for Developing Reliable Web Services

Table 5–2 provides best practices for developing reliable Web services. For best practices when accessing reliable Web services from behind a firewall, see Section 5.3, "Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection)."

Table 5–2 Roadmap for Developing Reliable Web Services

Best Practice	Description
Set the base retransmission interval to a realistic value for your particular scenario.	For more information, see Section 6.7.4, "Configuring the Base Retransmission Interval."
Set the acknowledgement interval to a realistic value for your particular scenario.	The recommended setting is two times the nominal interval between requests. For more information, see Section 6.7.9, "Configuring the Acknowledgement Interval."
Set timeouts (inactivity and sequence expiration) to realistic values for your particular scenario.	Consider the following: ■ For very short-lived exchanges, the default timeouts may be too long and sequence state might be maintained longer than necessary. ■ Set timeouts to two times the expected lifetime of a given business unit of work. This allows the sequence to live long enough For more information, see Section 6.7.7, "Configuring Inactivity Timeout" and Section 6.7.6, "Configuring the Sequence Expiration."
Use an reliable messaging policy that reflects the minimum delivery assurance (or quality of service) required.	By default, the delivery assurance is set to Exactly Once, In Order. If you do not require ordering, it can increase performance to set the delivery assurance to simply Exactly Once. Similarly, if your service can tolerate duplicate requests, delivery assurance can be set to At Least Once. For more information about delivery assurance for reliable messaging, see Table 6–1, " Delivery Assurances for Reliable Messaging" and Section 6.4, "Creating the Web Service Reliable Messaging WS-Policy File."

5.3 Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection)

Table 5–3 provides best practices for accessing reliable Web services from behind a firewall using Make Connection. These guidelines should be used in conjunction with the general guidelines provided in Section 5.2, "Roadmap for Developing Reliable Web Services" and Section 3.2, "Roadmap for Developing Asynchronous Web Service Clients."

Table 5–3 Roadmap for Accessing Reliable Web Services from Behind a Firewall (Make Connection)

Best Practice	Description
Coordinate the Make Connection polling interval with the reliable messaging base retransmission interval.	The polling interval you set for Make Connection transport sets the lower limit for the amount of time it takes for reliable messaging protocol messages to make the round trip between the client and service. If you set the reliable messaging base retransmission interval to a value near to the Make Connection polling interval, it will be unlikely that a reliable messaging request will be received by the Web service, and the accompanying RM acknowledgement sent for that request (at best one Make Connection polling interval later) before the reliable messaging runtime attempts to retransmit the request. Setting the reliable messaging base retransmission interval to a value that is too low results in unnecessary retransmissions for requests, and potentially a cascading load on the service side as it attempts to process redundant incoming requests and Make Connection poll messages to retrieve the responses from those requests. Oracle recommends setting the base retransmission interval to a value that is at least two times the Make Connection polling interval. **Note**: When Web services reliable messaging and Make Connection are used together, the Make Connection polling interval value will be adjusted at runtime, if necessary, to ensure that the value is set at least 3 seconds less than the reliable messaging base transmission interval. If the base transmission interval is three seconds or less, the Make Connection polling interval is set to the value of the base retransmission interval. For more information setting the Make Connection polling interval and reliable messaging base retransmission interval, see Section 4.6.2.2, "Configuring the Polling Interval" and Section 6.7.4, "Configuring the Base Retransmission Interval", respectively.

5.4 Roadmap for Securing Reliable Web Services

Table 5–4 provides best practices for securing reliable Web services using WS-SecureConversation. These guidelines should be used in conjunction with the guidelines provided in Section 5.2, "Roadmap for Developing Reliable Web Services."

Table 5–4 Roadmap for Securing Reliable Web Services

Best Practice	Description
Coordinate the WS-SecureConversation lifetime with the reliable messaging base retransmission and acknowledgement intervals.	A WS-SecureConversation lifetime that is set to a value near to or less than the reliable messaging base retransmission and acknowledgement intervals may result in the WS-SecureConversation token expiring before the reliable messaging handshake message can be sent to the Web service. For this reason, Oracle recommends setting the WS-SecureConversation lifetime to a value that is at least two times the base retransmission interval. For more information setting the base retransmission interval, see Section 6.7.4, "Configuring the Base Retransmission Interval."

6

Using Web Services Reliable Messaging

This chapter describes how to use Web services reliable messaging (WS-ReliableMessaging) for WebLogic Web services using Java API for XML Web Services (JAX-WS).

See also Section 5, "Roadmap for Developing Reliable Web Services and Clients".

This chapter includes the following sections:

- Section 6.1, "Overview of Web Services Reliable Messaging"
- Section 6.2, "Steps to Create and Invoke a Reliable Web Service"
- Section 6.3, "Configuring the Source and Destination WebLogic Server Instances"
- Section 6.4, "Creating the Web Service Reliable Messaging WS-Policy File"
- Section 6.5, "Programming Guidelines for the Reliable JWS File"
- Section 6.6, "Invoking a Reliable Web Service from a Web Service Client"
- Section 6.7, "Configuring Reliable Messaging"
- Section 6.8, "Implementing the Reliability Error Listener"
- Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence"
- Section 6.10, "Monitoring Web Services Reliable Messaging"
- Section 6.11, "Grouping Messages into Business Units of Work (Batching)"
- Section 6.12, "Client Considerations When Redeploying a Reliable Web Service"
- Section 6.13, "Interoperability with WebLogic Web Service Reliable Messaging"

The WebLogic Server Examples Server includes three reliable messaging examples:

- Configuring Reliable Messaging for JAX-WS Web Services
- Using Make Connection and Reliable Messaging for JAX-WS Web Services
- Configuring Secure and Reliable Messaging for JAX-WS Web Services

For more information, see "Web Services Samples in the WebLogic Serer Distribution" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

6.1 Overview of Web Services Reliable Messaging

Web service reliable messaging is a framework that enables an application running on one application server to *reliably* invoke a Web service running on another application server, assuming that both servers implement the WS-ReliableMessaging specification. Reliable is defined as the ability to guarantee message delivery between the two

endpoints (Web service and client) in the presence of software component, system, or network failures.

WebLogic Web services conform to the WS-ReliableMessaging 1.2 specification (February 2009) at http://docs.oasis-open.org/ws-rx/wsrm/200702 (and supports version 1.1). This specification describes how two endpoints (Web service and client) on different application servers can communicate reliably. In particular, the specification describes an interoperable protocol in which a message sent from a *source endpoint* (or client Web service) to a *destination endpoint* (or Web service whose operations can be invoked reliably) is guaranteed either to be delivered, according to one or more *delivery assurances*, or to raise an error.

A reliable WebLogic Web service provides the following delivery assurances.

Table 6–1 Delivery Assurances for Reliable Messaging

Delivery Assurance	Description
At Most Once	Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
At Least Once	Every message is delivered at least once. It is possible that some messages are delivered more than once.
Exactly Once	Every message is delivered exactly once, without duplication.
In Order	Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances.

The following sections describe how to create reliable Web services and clients and how to configure WebLogic Server instances to which the Web services are deployed.

6.1.1 Using WS-Policy to Specify Reliable Messaging Policy Assertions

WebLogic Web services use WS-Policy files to enable a destination endpoint to describe and advertise its Web service reliable messaging capabilities and requirements. The WS-Policy files are XML files that describe features such as the version of the supported WS-ReliableMessaging specification and quality of service requirements. The WS-Policy specification (http://www.w3.org/TR/ws-policy/) provides a general purpose model and syntax to describe and communicate the policies of a Web service.

WebLogic Server includes pre-packaged WS-Policy files that contain typical reliable messaging assertions, as described in Section A, "Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection". If the pre-packaged WS-Policy files do not suit your needs, you must create your own WS-Policy file. See Section 6.4, "Creating the Web Service Reliable Messaging WS-Policy File" for details. See "Web Service Reliable Messaging Policy Assertion Reference" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for reference information about the reliable messaging policy assertions.

6.1.2 Supported Transport Types for Reliable Messaging

You can use Web service reliable messaging asynchronously or synchronously. When delivering messages asynchronously, you can configure buffering to support automatic message delivery retries, if desired.

The following table summarizes the transport type support for Web services reliable messaging. For information about transport type support for Web service clients, see

Section 6.6, "Invoking a Reliable Web Service from a Web Service Client". For failure recovery information, see Section 6.1.4, "Reliable Messaging Failure Recovery Scenarios"

Note: Message buffering is configurable for Web services, as described in Chapter 8, "Configuring Message Buffering for Web Services.". For Web service clients, message buffering is enabled by default.

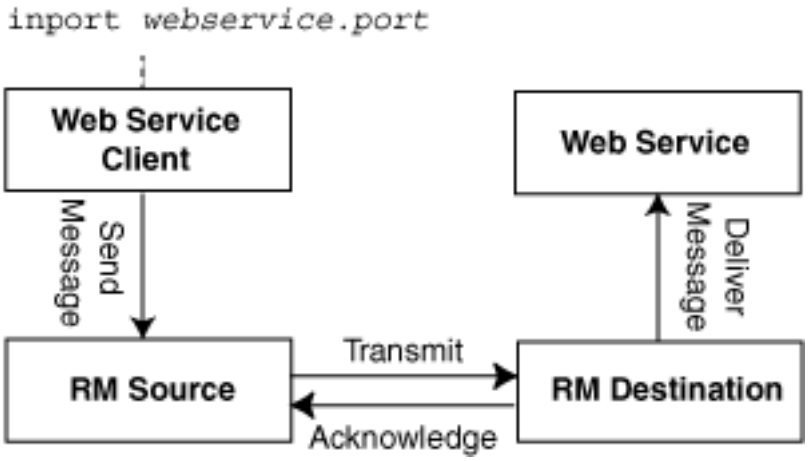
Table 6–2 Transport Types for Web Services Reliable Messaging

Transport Type	Features
Asynchronous transport	For buffered Web services: ■ Most robust usage mode, but requires the most overhead. ■ Automatically retries message delivery. ■ Survives network outages. ■ Enables restart of the source or destination endpoint. ■ Uses non-anonymous ReplyTo. ■ Employs asynchronous client transport enabling a single thread to service multiple requests, absorbing load more efficiently. For more information, see Section 4.5, "Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)." ■ Web service clients can use asynchronous or synchronous invocation semantics to invoke the Web service. For more information, see Table 4–1, " Support for Asynchronous Web Service Invocation". For non-buffered Web services: ■ Less overhead than asynchronous, buffered usage mode. ■ Persists sequence state only. ■ Uses non-anonymous ReplyTo. ■ Web service clients can use asynchronous or synchronous invocation semantics to invoke the Web service. For more information, see Table 4–1, " Support for Asynchronous Web Service Invocation".
Synchronous transport	■ Offers the least overhead and simplest programming model. ■ Uses anonymous ReplyTo. ■ Web service clients can use asynchronous or synchronous invocation semantics to invoke the Web service. For more information, see Table 4–1, " Support for Asynchronous Web Service Invocation". ■ If a Web service client invokes a buffered Web service using synchronous transport, one of following will result: - If this is the first request of the sequence, the destination sequence will be set to be non-buffered (as though the Web service configuration was set as non-buffered). - If this is not the first request of the sequence (that is, the client sent a request using asynchronous transport previously), then the request is rejected and a fault returned.

6.1.3 The Life Cycle of the Reliable Message Sequence

The following figure shows a one-way reliable message exchange.

Figure 6–1 Web Service Reliable Message Exchange



A *reliable message sequence* is used to track the progress of a set of messages that are exchanged reliably between an RM source and RM destination. A sequence can be used to send zero or more messages, and is identified by a string *identifier*. This identifier is used to reference the sequence when using reliable messaging.

The Web service client application sends a message for reliable delivery which is transmitted by the RM source to the RM destination. The RM destination acknowledges that the reliable message has been received and delivers it to the Web service application. The message may be retransmitted by the RM source until the acknowledgement is received. The RM destination, if configured to buffer requests, may redeliver the request to the Web service if the Web service fails to process the request.

A Web service client sends messages to a target Web service by invoking methods on the client instance (port or Dispatch instance). A *port* is associated with the port type of the reliable Web service and represents a programmatic interface to that service. The port is created by the `<clientgen>` child element of the `jwsc` Ant task. A *Dispatch instance* is a loosely-typed, general-purpose interface for delivering whole messages from the client to the Web service. For more information about Dispatch clients, see Section 19.6, "Developing a Web Service Dispatch Client.".

WebLogic stores the identifier for the reliable message sequence within this client instance. This causes the reliable message sequence to be connected to a single client instance. All messages that are sent using a given client instance will use the same reliable messaging sequence, regardless of the number of messages that are sent. (Unless you using batching, as described in Section 6.11, "Grouping Messages into Business Units of Work (Batching).")

Because WebLogic Server retains resources associated with the reliable sequence, it is recommended that you take steps to release these resources in a timely fashion. this can be done by managing the lifecycle of the client instance itself, or by using the `weblogic.wsee.reliability2.api.WsrmClient` API. Use the `WsrmClient` API to perform common tasks such as set configuration options, get the sequence id, and terminate a reliable sequence. For more information, see Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence."

6.1.4 Reliable Messaging Failure Recovery Scenarios

The following sections outline reliable messaging failure recovery for various scenarios.

- Section 6.1.4.1, "RM Destination Down Before Request Arrives"
- Section 6.1.4.2, "RM Source Down After Request is Made"
- Section 6.1.4.3, "RM Destination Down After Request Arrives"
- Section 6.1.4.4, "Failure Scenarios with Non-buffered Reliable Web Services"

The first three scenarios assume that buffering is enabled on both the Web service and client. The last scenario describes reliable messaging failure recovery for non-buffered Web services. Buffering is enabled on Web service client by default. To configure buffering on the Web service, see Chapter 8, "Configuring Message Buffering for Web Services."

6.1.4.1 RM Destination Down Before Request Arrives

Table 6–3 describes the reliable messaging failure recovery scenario when an RM destination is unavailable before a request from the RM source arrives.

It is assumed that Web service buffering is enabled on both the Web service and client. Buffering is enabled on Web service client by default. To configure buffering on the Web service, see Chapter 8, "Configuring Message Buffering for Web Services."

Table 6–3 Reliable Messaging Failure Recovery Scenario—RM Destination Down Before Request Arrives

Transport Type	Scenario Description
Asynchronous Transport	1. Client invokes an asynchronous method. 2. Reliable messaging runtime accepts the request; client returns to do other work. 3. Reliable messaging runtime attempts to deliver the request and fails because the RM destination is down. 4. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery fails again. 5. RM destination comes up. 6. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery succeeds. 7. Acknowledgement is sent to the client which includes the message number of the request. The reliable messaging runtime removes the message from the retry list. 8. Response arrives and the client processes it. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence".

Table 6–3 (Cont.) Reliable Messaging Failure Recovery Scenario—RM Destination Down Before Request

Transport Type	Scenario Description
Synchronous Transport	1. Client invokes a synchronous method. 2. Reliable messaging runtime accepts the request and blocks the client thread. 3. Reliable messaging runtime attempts to deliver the request and fails because the RM destination is down. 4. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery fails again. 5. RM destination comes up. 6. Reliable messaging runtime waits for the retry interval and tries to send the request again. The request delivery succeeds. 7. Response and acknowledgement are sent to the client via the transport back-channel. The acknowledgement includes the message number of the request. The reliable messaging runtime removes the message from the retry list. 8. Reliable messaging runtime unblocks the client thread and returns the response. 9. Client receives the response as the return value of the method invocation, and processes the response. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence". **Note**: To achieve true reliability with synchronous transport, it is recommended that you use Make Connection. For more information, see Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)."

6.1.4.2 RM Source Down After Request is Made

Table 6–4 describes the reliable messaging failure recovery scenario when an RM source goes down after a request is made.

It is assumed that Web service buffering is enabled on both the Web service and client. Buffering is enabled on Web service clients by default. To configure buffering on the Web service, see Chapter 8, "Configuring Message Buffering for Web Services."

Table 6–4 Reliable Messaging Failure Recovery Scenario—RM Source Down After Request is Made

Transport Type	Scenario Description
Asynchronous Transport	1. Client invokes an asynchronous method. 2. Reliable messaging runtime accepts the request; client returns to do other work. 3. Client (RM source) goes down. 4. Client comes up. Client must re-initialize the client instance using the same client ID. The runtime will use this client ID to retrieve the reliable sequence ID that was active for the client. For more information, see Section 6.9.2, "Managing the Client ID". 5. Reliable messaging runtime detects the reliable sequence ID that was in use prior to the client going down and recovers the accepted requests. **Note**: This step is accomplished only after the client re-initializes the client instance that was used to send the request because delivery of the request depends on resources provided by the client instance. It is recommended that clients initialize the client instance in a static block, or use a `@PostConstruct` annotation or other mechanism to ensure early initialization of the client instance. For more information, see the best practices examples presented in Section 3, "Roadmaps for Developing Web Service Clients." 6. Reliable messaging runtime sends the request and succeeds. 7. Acknowledgement is sent to the client which includes the message number of the request. The reliable messaging runtime removes the message from the retry list. 8. Response arrives and the client processes it. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence".

Table 6–4 (Cont.) Reliable Messaging Failure Recovery Scenario—RM Source Down After Request is Made

Transport Type	Scenario Description
Synchronous Transport	1. Client invokes a synchronous method. 2. Reliable messaging runtime accepts the request and blocks the client thread. 3. Reliable messaging runtime attempts to deliver the request. The request delivery succeeds. 4. Before response can be sent, the client (RM source) goes down. Client thread is lost as the VM exits, along with the invocation state and calling stack of the client itself. 5. Client (RM source) comes up. Client must re-initialize the client instance (port or Dispatch) using the same client ID. For more information, see Section 6.9.2, "Managing the Client ID" 6. Reliable messaging runtime detects the previous sequence ID for the client, and sees that the last request was made synchronously. 7. Reliable messaging runtime delivers a permanent failure notification for this request, and fails the entire RM sequence associated with the client instance. Any `ReliabilityErrorListener` associated with the client instance will be called at this point. 8. Client is responsible for retrieving the original request (via some client-specific mechanism) and resending it by re-invoking the client instance with the request. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence". **Note**: To achieve true reliability with synchronous transport, it is recommended that you use Make Connection. For more information, see Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)."

6.1.4.3 RM Destination Down After Request Arrives

Table 6–5 describes the reliable messaging failure recovery scenario when an RM destination is unavailable after a request has been accepted from the RM source.

It is assumed that Web service buffering is enabled on both the Web service and client. Buffering is enabled on Web service client by default. To configure buffering on the Web service, see Chapter 8, "Configuring Message Buffering for Web Services."

Table 6–5 Reliable Messaging Failure Recovery Scenario—RM Destination Down After Request Arrives

Transport Type	Scenario Description
Asynchronous Transport	1. Client invokes an asynchronous method. 2. Reliable messaging runtime accepts the request; client returns to do other work. 3. Reliable messaging runtime attempts to deliver the request and succeeds. 4. The RM destination accepts the request and send an acknowledgement on the back channel. 5. Reliable messaging runtime sees the acknowledgement and removes the message from the retry list. 6. RM destination goes down. 7. Reliable messaging runtime on RM source retries any pending requests during this time. 8. RM destination comes up. 9. RM destination recovers the stored request, processes it, and sends the response. 10. Response arrives and the client processes it. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence".
Synchronous Transport	**Note**: If you attempt to invoke a buffered Web service using synchronous transport, one of following will result: ▪ If this is the first request of the sequence, the destination sequence will be set to be non-buffered (as though the Web service configuration was set as non-buffered). ▪ If this is not the first request of the sequence (that is, the client sent a request using asynchronous transport previously), then the request is rejected and a fault returned. The following describes the sequence of this scenario: 1. Client invokes a synchronous method. 2. Reliable messaging runtime accepts the request and blocks the client thread. 3. Reliable messaging runtime attempts to deliver the request. The request delivery succeeds. 4. RM destination accepts the request and sends an acknowledgement via the transport back channel. 5. Client (RM source) detects the acknowledgement and removes the request from the retry list. 6. RM destination goes down. 7. Client thread remains blocked. 8. RM Destination comes up, recovers, and processes the request, and sends the response to the client. 9. Reliable messaging runtime unblocks the client thread and returns the response. 10. Client receives the response as the return value of the method invocation, and processes the response. **Note**: At any time, the client can check acknowledgement status, access information about a message, and so on, as described in Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence".

6.1.4.4 Failure Scenarios with Non-buffered Reliable Web Services

A non-buffered Web service operates differently than a buffered Web service in that it does not buffer a request to hardened storage before acknowledging it and attempting to process it. A non-buffered Web service will not attempt to reprocess a request if the service logic fails, whereas a buffered Web service will attempt to reprocess the request. In both cases, buffered or non-buffered, any response generated by the Web service will be buffered before it is sent back to the client.

A non-buffered Web service may be useful in the following cases:

- Web service operates against non-transactional resources and should not process any request more than once (because rolling back the transaction that dequeued the buffered request cannot roll back the side effects of the non-transactional service).
- Web service is relatively light weight, and does not take very long to process requests.
- Web service performance is of paramount importance and risk of losing request or response is acceptable. Non-buffered Web services will not incur the overhead of buffering the request to a store, and thus can deliver better throughput than a buffered Web service. The performance gain is dependent on how much time and resources are required to buffer the requests (for example, very large request messages may take significant time and resources to buffer).

A non-buffered Web service is operationally similar to a buffered Web service in most failure scenarios. The exceptions are cases where the service (RM destination) itself fails. For example, in all the RM source failure scenarios described, the behavior is the same for a buffered or a non-buffered Web service (RM destination). For non-buffered Web services the failure window is open between the following two points:

- The request is accepted for processing.
- The response from the Web service is registered for delivery to the client (RM source).

If the Web service (RM destination) fails between these two points, the RM source will assume the request has been successfully processed (since it has been acknowledged) but will *never* receive a response, and the request may never have been processed.

Carefully consider this failure window before configuring a Web service to run as non-buffered.

6.2 Steps to Create and Invoke a Reliable Web Service

Configuring reliable messaging for a WebLogic Web service requires standard JMS tasks such as creating JMS servers and Store and Forward (SAF) agents, as well as Web service-specific tasks, such as adding additional JWS annotations to your JWS file. Optionally, you create custom WS-Policy files that describe the reliable messaging capabilities of the reliable Web service if you do not use the pre-packaged ones.

If you are using the WebLogic client APIs to invoke a reliable Web service, the client application must run on WebLogic Server. Thus, configuration tasks must be performed on both the *source* WebLogic Server instance on which the Web service client code is deployed, as well as the *destination* WebLogic Server instance on which the reliable Web service itself is deployed.

Table 6–6 summarizes the steps to create a reliable Web service and a client that invokes an operation of the reliable Web service. The procedure describes how to create the JWS files that implement the Web service and client from scratch; if you

want to update existing JWS files, use this procedure as a guide. The procedure also describes how to configure the source and destination WebLogic Server instances.

It is assumed that you have completed the following tasks:

- You have created the *destination* and *source* WebLogic Server instances. You deploy the reliable Web service to the *destination* WebLogic Server instance, and the client that invokes the reliable Web service to the *source* WebLogic Server instance.
- You have set up an Ant-based development environment.
- You have working `build.xml` files that you can edit, for example, to add targets for running the `jwsc` Ant task and deploying the generated reliable Web service.

For more information, see "Developing WebLogic Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*. For best practices for developing asynchronous and reliable Web services and clients, see Section 5, "Roadmap for Developing Reliable Web Services and Clients".

Table 6–6 Steps to Create and Invoke a Reliable Web Service

#	Step	Description
1	Configure the *destination* and *source* WebLogic Server instances.	You deploy the reliable Web service to the *destination* WebLogic Server instance, and the client that invokes the reliable Web service to the *source* WebLogic Server instance. For information about configuring the destination WebLogic Server instance, see Section 6.3, "Configuring the Source and Destination WebLogic Server Instances."
2	Create the WS-Policy file. (Optional)	Using your favorite XML or plain text editor, optionally create a WS-Policy file that describes the reliable messaging capabilities of the Web service running on the destination WebLogic Server. For details about creating your own WS-Policy file, see Section 6.4, "Creating the Web Service Reliable Messaging WS-Policy File." **Note**: This step is not required if you plan to use one of the WS-Policy files that are included in WebLogic Server; see Appendix A, "Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection" for more information.
3	Create or update the JWS file that implements the reliable Web service.	This Web service will be deployed to the destination WebLogic Server instance. See Section 6.5, "Programming Guidelines for the Reliable JWS File." For examples demonstrating best practices, see Section 5, "Roadmap for Developing Reliable Web Services and Clients."
4	Update the `build.xml` file that is used to compile the reliable Web services.	Update your `build.xml file` to include a call to the `jwsc` Ant task which will compile the reliable JWS file into a Web service. See "Running the jwsc WebLogic Web Services Ant Task" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* for general information about using the `jwsc` task.
5	Compile and deploy the reliable JWS file.	Compile the reliable JWS file by calling the appropriate target and deploy to the destination WebLogic Server. For example: `prompt> ant build-reliableService deploy-reliableService`
6	Create or update the Web service client.	The Web service client invokes the reliable Web service and will be deployed to the source WebLogic Server. See Section 6.6, "Invoking a Reliable Web Service from a Web Service Client".
7	Configure reliable messaging. (Optional)	Configure reliable messaging for the reliable Web service using the Administration Console. The WS-Policy file attached to the reliable Web service provides the initial configuration settings. See Section 6.7, "Configuring Reliable Messaging".
8	Implement a reliability error listener. (Optional)	Implement a reliability error listener to receive notifications if a reliable delivery fails. See Section 6.8, "Implementing the Reliability Error Listener".

Table 6–6 (Cont.) Steps to Create and Invoke a Reliable Web Service

#	Step	Description
9	Manage the life cycle of a reliable message sequence. (Optional)	WebLogic Server provides a client API, `weblogic.wsee.reliability2.api.WsrmClient`, for use with the Web service reliable messaging. Use this API to perform common life cycle tasks such as set configuration options, get the reliable sequence id, and terminate a reliable sequence. See Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence".
10	Update the `build.xml` file that is used to compile the client Web service.	Update your `build.xml file` to include a call to the `jwsc` Ant task which will compile the reliable JWS file into a Web service. See "Running the jwsc WebLogic Web Services Ant Task" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* for general information about using the `jwsc` task.
11	Compile and deploy the Web service client file.	Compile your client file by calling the appropriate target and deploy to the source WebLogic Server. For example: `prompt> ant build-clientService deploy-clientService`
12	Monitor Web services reliable messaging.	Use the Administration Console to monitor Web services reliable messaging. See Section 6.10, "Monitoring Web Services Reliable Messaging".

Each of these steps is described in more detail in the following sections. In addition, the following topics are discussed:

- Section 6.11, "Grouping Messages into Business Units of Work (Batching)"—Describes how to group messages into business *units of work*—also called batching—to improve performance when using reliable messaging.
- Section 6.12, "Client Considerations When Redeploying a Reliable Web Service"—Describes client considerations for when you deploy a new version of an updated reliable WebLogic Web service alongside an older version of the same Web service.
- Section 6.13, "Interoperability with WebLogic Web Service Reliable Messaging"—Provides recommendations for interoperating with WebLogic Web services reliable messaging.

6.3 Configuring the Source and Destination WebLogic Server Instances

You need to configure Web service persistence on the destination and source WebLogic Server instances. You deploy the reliable Web service to the *destination* WebLogic Server instance, and the client that invokes the reliable Web service to the *source* WebLogic Server instance.

When using Web services reliable messaging, the Web services reliable messaging sequence is saved to the Web service persistent store any time its state changes. Examples of state change include:

- Reliable messaging state is updated (creating, created, terminating, terminated, and so on).
- Security property is updated (such as security context token)
- Message is sent on the reliable messaging sequence (if message buffering is enabled)
- Acknowledgement when a message arrives

You can configure Web service persistence using the Configuration Wizard to extend the WebLogic Server domain using a Web services-specific extension template.

Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Administration Console or WLST. For information about configuring Web service persistence, see Section 7.3, "Configuring Web Service Persistence."

You may also wish to configure buffering for Web services. For considerations and steps to configure message buffering, see Chapter 8, "Configuring Message Buffering for Web Services."

6.4 Creating the Web Service Reliable Messaging WS-Policy File

A WS-Policy file is an XML file that contains policy assertions that comply with the WS-Policy specification. In this case, the WS-Policy file contains Web service reliable messaging policy assertions.

WebLogic Server includes pre-packaged WS-Policy files that contain typical reliable messaging assertions that you can use if you do not want to create your own WS-Policy file.

The pre-packaged WS-Policy files are listed in the following table. This table also specifies whether the WS-Policy file can be attached at the method level; if the value in this column is no, then the WS-Policy file can be attached at the class level only. For more information, see Appendix A, "Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection"

Note: The `DefaultReliability.xml` and `LongRunningReliability.xml` files are deprecated in this release. Use of the `DefaultReliability1.2.xml`, `Reliability1.2_SequenceTransportSecurity`, or `Reliability1.0_1.2.xml` file is recommended and required to comply with the 1.2 version of the WS-ReliableMessaging specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.pdf`.

Table 6–7 Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`DefaultReliability1.2.xml`	Specifies policy assertions related to delivery assurance. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Section A.2, "DefaultReliability1.1.xml (WS-Policy File)".	Yes
`DefaultReliability1.1.xml`	Specifies policy assertions related to quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Section A.2, "DefaultReliability1.1.xml (WS-Policy File)".	Yes
`Reliability1.2_ExactlyOnce_WithMC1.1.xml`	Specifies policy assertions related to quality of service. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client. See Section A.7, "Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File)".	No

Table 6–7 (Cont.) Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`Reliability1.2_SequenceSTRSecurity`	Specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Section A.10, "Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File)".	No
`Reliability1.1_SequenceSTRSecurity`	The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Section A.11, "Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)"	Yes
`Reliability1.2_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Section A.10, "Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File)".	Yes
`Reliability1.1_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Section A.11, "Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)".	Yes
`Reliability1.0_1.2.xml`	Combines 1.2 and 1.0 WS-Reliable Messaging policy assertions. The policy assertions for the 1.2 version Make Connection support on the Web service and specifies usage as optional on the Web service client. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. See Section A.12, "Reliability1.0_1.2.xml (WS-Policy File)".	No

Table 6–7 (Cont.) Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`Reliability1.0_1.1.xml`	Combines 1.1 and 1.0 WS Reliable Messaging policy assertions. See Section A.13, "Reliability1.0_1.1.xml (WS-Policy.xml File)".	Yes
`DefaultReliability.xml`	**Deprecated.** The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf`. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. Specifies typical values for the reliable messaging policy assertions, such as inactivity timeout of 10 minutes, acknowledgement interval of 200 milliseconds, and base retransmission interval of 3 seconds. See Section A.3, "DefaultReliability.xml WS-Policy File (WS-Policy) [Deprecated]".	Yes
`LongRunningReliability.xml`	**Deprecated.** The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 for long running processes. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. Similar to the preceding default reliable messaging WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours.) See Section A.4, "LongRunningReliability.xml WS-Policy File (WS-Policy) [Deprecated]".	Yes

You can use one of the pre-packaged reliable messaging WS-Policy files included in WebLogic Server; these files are adequate for most use cases. You cannot modify the pre-packaged files. If the values do not suit your needs, you must create a custom WS-Policy file. The following sections describe how to create a custom WS-Policy file.

- Section 6.4.1, "Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Versions 1.2 and 1.1"
- Section 6.4.2, "Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Version 1.0 (Deprecated)"
- Section 6.4.3, "Using Multiple Policy Alternatives"

6.4.1 Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Versions 1.2 and 1.1

This section describes how to create a custom WS-Policy file that contains Web service reliable messaging assertions that are based on the following specifications:

- WS Reliable Messaging Policy Assertion Version 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.2-spec-os.html`
- WS Reliable Messaging Policy Assertion Version 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declaration:

```
<wsp:Policy
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
```

You wrap all Web service reliable messaging policy assertions inside of a `<wsrmp:RMAssertion>` element. This element should include the following namespace declaration for using Web service reliable messaging policy assertions:

```
<wsrmp:RMAssertion
   xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
```

The following table lists the Web service reliable messaging assertions that you can specify in the WS-Policy file. The order in which the assertions appear is important. You can specify the following assertions; the order they appear in the following list is the order in which they should appear in your WS-Policy file:

Table 6–8 Web Service Reliable Messaging Assertions (Versions 1.2 and 1.1)

Assertion	Description
`<wsrmp:SequenceSTR>`	To secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. You can only specify one security assertion; that is, you can specify `wsrmp:SequenceSTR` or `wsrmp:SequenceTransportSecurity`, but not both.
`<wsrmp:SequenceTransportSecurity>`	To secure messages in a reliable sequence, the runtime will use the SSL transport session that is used to send the `CreateSequence` message. This assertion must be used in conjunction with the `sp:TransportBinding` assertion that requires the use of some transport-level security mechanism (for example, `sp:HttpsToken`). You can only specify one security assertion; that is, you can specify `wsrmp:SequenceSTR` or `wsrmp:SequenceTransportSecurity`, but not both.
`<wsrm:DeliveryAssurance>`	Delivery assurance (or quality of service) of the Web service. Valid values are `AtMostOnce`, `AtLeastOnce`, `ExactlyOnce`, and `InOrder`. You can set one of the delivery assurances defined in the following table. If not set, the delivery assurance defaults to `ExactlyOnce`. For more information about delivery assurance, see Table 6–1.

The following example shows a simple Web service reliable messaging WS-Policy file:

```
<?xml version="1.0"?>

<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceTransportSecurity/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

For more information about Reliable Messaging policy assertions in the WS-Policy file, see "Web Service Reliable Messaging Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

6.4.2 Creating a Custom WS-Policy File Using WS-ReliableMessaging Policy Assertions Version 1.0 (Deprecated)

This section describes how to create a custom WS-Policy file that contains Web service reliable messaging assertions that are based on WS Reliable Messaging Policy Assertion Version 1.0 at http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf.

> **Note:** Many of the reliable messaging policy assertions described in this section are managed through JWS annotations or configuration.

The root element of the WS-Policy file is `<Policy>` and it should include the following namespace declarations for using Web service reliable messaging policy assertions:

```
<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy">
```

You wrap all Web service reliable messaging policy assertions inside of a `<wsrm:RMAssertion>` element. The assertions that use the `wsrm:` namespace are standard ones defined by the WS-ReliableMessaging specification at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf. The assertions that use the `beapolicy:` namespace are WebLogic-specific. See "Web Service Reliable Messaging Policy Assertion Reference" in the *WebLogic Web Services Reference for Oracle WebLogic Server* for details.

The following table lists the Web service reliable messaging assertions that you can specify in the WS-Policy file. All Web service reliable messaging assertions are optional, so only set those whose default values are not adequate. The order in which the assertions appear is important. You can specify the following assertions; the order they appear in the following list is the order in which they should appear in your WS-Policy file,

Table 6–9 Web Service Reliable Messaging Assertions (Version 1.0)

Assertion	Description
`<wsrm:InactivityTimeout>`	Number of milliseconds, specified with the `Milliseconds` attribute, which defines an inactivity interval. After this amount of time, if the destination endpoint has not received a message from the source endpoint, the destination endpoint may consider the sequence to have terminated due to inactivity. The same is true for the source endpoint. By default, sequences never timeout.
`<wsrm:BaseRetransmissionInterval>`	Interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message if it receives no acknowledgment for that message. Default value is set by the SAF agent on the source endpoint's WebLogic Server instance.
`<wsrm:ExponentialBackoff>`	Specifies that the retransmission interval will be adjusted using the exponential backoff algorithm. This element has no attributes.

Table 6–9 (Cont.) Web Service Reliable Messaging Assertions (Version 1.0)

Assertion	Description
`<wsrm:AcknowledgmentInterval>`	Maximum interval, in milliseconds, in which the destination endpoint must transmit a stand-alone acknowledgement. The default value is set by the SAF agent on the destination endpoint's WebLogic Server instance.
`<beapolicy:Expires>`	Amount of time after which the reliable Web service expires and does not accept any new sequence messages. The default value is to never expire. This element has a single attribute, `Expires`, whose data type is an XML Schema duration type (see `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#duration`). For example, if you want to set the expiration time to one day, use the following: `<beapolicy:Expires Expires="P1D" />`.
`<beapolicy:QOS>`	Delivery assurance level, as described in Table 6–1. The element has one attribute, `QOS`, which you set to one of the following values: `AtMostOnce`, `AtLeastOnce`, or `ExactlyOnce`. You can also include the `InOrder` string to specify that the messages be in order. The default value is `ExactlyOnce InOrder`. This element is typically not set.

The following example shows a simple Web service reliable messaging WS-Policy file:

```
<?xml version="1.0"?>

<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy"
  >
 <wsrm:RMAssertion>
   <wsrm:InactivityTimeout
      Milliseconds="600000" />
   <wsrm:BaseRetransmissionInterval
      Milliseconds="500" />
   <wsrm:ExponentialBackoff />
   <wsrm:AcknowledgementInterval
      Milliseconds="2000" />
 </wsrm:RMAssertion>
</wsp:Policy>
```

For more information about reliable messaging policy assertions in the WS-Policy file, see "Web Service Reliable Messaging Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

6.4.3 Using Multiple Policy Alternatives

You can configure multiple policy alternatives—also referred to as *smart policy alternatives*—for a single Web service by creating a custom policy file. At runtime, WebLogic Server selects which of the configured policies to apply. It excludes policies that are not supported or have conflicting assertions and selects the appropriate policy, based on your configured preferences, to verify incoming messages and build the response messages.

The following example provides an example of a security policy that supports both 1.2 and 1.0 WS-Reliable Messaging. Each policy alternative is enclosed in a `<wsp:All>` element.

Note: The 1.0 Web service reliable messaging assertions are prefixed by `wsrmp10`.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsrmp10:RMAssertion
       xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp10:InactivityTimeout Milliseconds="1200000"/>
        <wsrmp10:BaseRetransmissionInterval Milliseconds="60000"/>
        <wsrmp10:ExponentialBackoff/>
        <wsrmp10:AcknowledgementInterval Milliseconds="800"/>
      </wsrmp10:RMAssertion>
    </wsp:All>
    <wsp:All>
      <wsrmp:RMAssertion
           xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
        <wsrmp:SequenceSTR/>
        <wsrmp:DeliveryAssurance>
          <wsp:Policy>
            <wsrmp:AtMostOnce/>
          </wsp:Policy>
        </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

For more information about multiple policy alternatives, see "Smart Policy Selection" in *Securing WebLogic Web Services for Oracle WebLogic Server*.

6.5 Programming Guidelines for the Reliable JWS File

Note: For best practices for developing reliable Web services, see Chapter 5, "Roadmap for Developing Reliable Web Services and Clients."

Use the `@Policy` annotation in your JWS file to specify that the Web service has a WS-Policy file attached to it that contains reliable messaging assertions. WebLogic Server delivers a set of pre-packaged WS-Policy files, as described in Appendix A, "Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection".

Follow the following guidelines when using the `@Policy` annotation for Web service reliable messaging:

- Use the `uri` attribute to specify the build-time location of the policy file, as follows:
  - If you have created your own WS-Policy file, specify its location relative to the JWS file. For example:

```
@Policy(uri="ReliableHelloWorldPolicy.xml",
          direction=Policy.Direction.both,
          attachToWsdl=true)
```

In this example, the `ReliableHelloWorldPolicy.xml` file is located in the same directory as the JWS file.

– To specify one of the pre-packaged WS-Policy files or a WS-Policy file that is packaged in a shared Java EE library, use the `policy:` prefix along with the name and path of the policy file. This syntax tells the `jwsc` Ant task at build-time *not* to look for an actual file on the file system, but rather, that the Web service will retrieve the WS-Policy file from WebLogic Server at the time the service is deployed.

Note: Shared Java EE libraries are useful when you want to share a WS-Policy file with multiple Web services that are packaged in different Enterprise applications. As long as the WS-Policy file is located in the `META-INF/policies` or `WEB-INF/policies` directory of the shared Java EE library, you can specify the policy file in the same way as if it were packaged in the same archive at the Web service. See "Creating Shared Java EE Libraries and Optional Packages" in *Developing Applications for Oracle WebLogic Server* for information about creating libraries and setting up your environment so the Web service can locate the policy files.

– To specify that the policy file is published on the Web, use the `http:` prefix along with the URL, as shown in the following example:

```
@Policy(uri="http://someSite.com/policies/mypolicy.xml"
        direction=Policy.Direction.both,
        attachToWsdl=true)
```

- By default, WS-Policy files are applied to both the request (inbound) and response (outbound) SOAP messages. You can change this default behavior with the `direction` attribute by setting the attribute to `Policy.Direction.inbound` or `Policy.Direction.outbound`.
- You can specify whether the Web service requires the operations to be invoked reliably and have the responses delivered reliably using the `wsp:optional` attribute within the policy file specified by `uri`.

  Please note:

  – If the client uses synchronous transport to invoke a Web service, and the inbound direction of the operation requires reliability (`optional` attribute is `false`), the client must provide an *offer sequence* (`<wsrm: Offer...>` as described in the WS-ReliableMessaging specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`) for use when sending reliable responses.

  – If the client uses asynchronous transport, the client is not required to send an offer sequence. If a request is made reliably, and the outbound direction has any RM policy (optional or not), the reliable messaging runtime will enforce the handshaking of a *new* RM sequence for sending the response. This new sequence will be associated with the request sequence, and all responses from that point onward are sent on the new response sequence. The response sequence is negotiated with the endpoint indicated by the ReplyTo address of the request.

- Set the `attachToWsdl` attribute of the `@Policy` annotation to specify whether the policy file should be attached to the WSDL file that describes the public contract of the Web service. Typically, you want to publicly publish the policy so

that client applications know the reliable messaging capabilities of the Web service. For this reason, the default value of this attribute is `true`.

For more information about the `@Policy` annotation, see "weblogic.jws.Policy" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

Example 6–1shows a simple JWS file that implements a reliable Web service.

Example 6–1 Example of a Reliable Web Service

```
import javax.jws.WebService;

import weblogic.jws.Policies;
import weblogic.jws.Policy;

/**
 * Example Web service for reliable client best practice examples
 */
@WebService
// Enable RM on this service.
@Policies( { @Policy(uri = "policy:DefaultReliability1.2.xml") })
public class BackendReliableService {

  public String doSomething(String what) {

    System.out.println("BackendReliableService doing: " + what);

    return "Did (Reliably) '" + what + "' at: " + System.currentTimeMillis();
  }
}
```

In the example, the predefined `DefaultReliability1.2.xml` policy file is attached to the Web service at the class level, which means that the policy file is applied to all public operations of the Web service—the `doSomething()` operation can be invoked reliably. The policy file is applied to both request and response by default. For information about the pre-packaged policies available and creating a custom policy, see Section 6.4, "Creating the Web Service Reliable Messaging WS-Policy File".

6.6 Invoking a Reliable Web Service from a Web Service Client

Note: For best practices for developing reliable Web service clients, see Section 5.1, "Roadmap for Developing Reliable Web Service Clients."

The following table summarizes how to invoke a reliable Web service from a Web service client based on the transport type that you want to employ. For a description of transport types, see Table 6–2.

Table 6–10 Invoking a Reliable Web Service Based on Transport Type

Transport Type	Description
Asynchronous transport	To use asynchronous transport, perform the following steps: **1.** Implement the Web service client, as described in Table 4–3, " Steps to Invoke Web Services Asynchronously". In step 3 of Table 4–3, implement one of the following transport mechanisms, depending on whether the client is behind a firewall or not: -Asynchronous client transport feature, as described in Section 4.5, "Developing Scalable Asynchronous JAX-WS Clients (Asynchronous Client Transport)". - Make Connection if the client is behind a firewall, as described in Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)". **2.** Invoke the Web service using either asynchronous or synchronous invocation semantics. **Note**: You can invoke synchronous operations when asynchronous client transport or Make Connection is enabled, as described in Section 4.5.1.5, "Configuring Asynchronous Client Transport for Synchronous Operations" and Section 4.6.2.4, "Configuring Make Connection as the Transport for Synchronous Methods".
Synchronous transport	To use synchronous transport, invoke an asynchronous or synchronous method on the reliable messaging service port instance using the standard JAX-WS Reference Implementation, as described in Section 4.7, "Using the JAX-WS Reference Implementation". **Note**: If you attempt to invoke a buffered Web service using synchronous transport, one of following will result: ▪ If this is the first request of the sequence, the destination sequence will be set to be non-buffered (as though the Web service configuration was set as non-buffered). ▪ If this is not the first request of the sequence (that is, the client sent a request using asynchronous transport previously), then the request is rejected and a fault returned.

For additional control on the client side, you may wish to perform one or more of the following tasks:

- Configure reliable messaging on the client side, as described in Section 6.7, "Configuring Reliable Messaging".
- Implement the reliability error listener to receive notifications if a reliable delivery fails, as described in Section 6.8, "Implementing the Reliability Error Listener". Oracle recommends that you always implement the reliability error listener as a best practice.
- Perform common life cycle tasks on the reliable messaging sequence, such as set configuration options, get the reliable sequence id, and terminate a reliable sequence, as described in Section 6.9, "Managing the Life Cycle of a Reliable Message Sequence"

6.7 Configuring Reliable Messaging

Note: For best practices for configuring reliable Web services, see Chapter 5, "Roadmap for Developing Reliable Web Services and Clients."

You can configure properties for a reliable Web service and client at the WebLogic Server, Web service endpoint, or Web service client level.

The properties that you define at the WebLogic Server level apply to all reliable Web services and clients on that server. For information about configuring reliable messaging at the WebLogic Server level, see Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server".

If desired, you can override the reliable message configuration options defined at the server level, as follows:

- At the Web service endpoint level by updating the application *deployment plan*. The deployment plan associates new values with specific locations in the descriptors for your application, and is stored in the `weblogic-webservices.xml` descriptor. At deployment time, a deployment plan is merged with the descriptors in the application by applying the values in its variable assignments to the locations in the application descriptors to which the variables are linked. For more information, see Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"
- At the Web service client level, as described in Section 6.7.3, "Configuring Reliable Messaging on Web Service Clients"

The following sections describe how to configure reliable messaging at the WebLogic Server, Web service endpoint, and Web service client levels.

- Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server"
- Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"
- Section 6.7.3, "Configuring Reliable Messaging on Web Service Clients"
- Section 6.7.4, "Configuring the Base Retransmission Interval"
- Section 6.7.5, "Configuring the Retransmission Exponential Backoff"
- Section 6.7.6, "Configuring the Sequence Expiration"
- Section 6.7.7, "Configuring Inactivity Timeout"
- Section 6.7.8, "Configuring a Non-buffered Destination for a Web Service"
- Section 6.7.9, "Configuring the Acknowledgement Interval"
- Section 6.8, "Implementing the Reliability Error Listener"

6.7.1 Configuring Reliable Messaging on WebLogic Server

You can configure reliable messaging on WebLogic Server using the Administration Console or WLST, as described in the following sections.

- Section 6.7.1.1, "Using the Administration Console"
- Section 6.7.1.2, "Using WLST"

6.7.1.1 Using the Administration Console

To configure reliable messaging for WebLogic Server using the Administration Console:

1. Invoke the Administration Console, as described in "Using the WebLogic Server Administration Console" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
2. In the left navigation pane, select **Environment**, then **Servers**.

3. Select the **Configuration** tab and in the Server tables, click on the name of the server for which you want to configure reliable messaging.
4. Click the **Configuration** tab, then the **Web Services** tab, then the **Reliable Message** tab.
5. Edit the reliable messaging properties, as described in the following sections:
   - Section 6.7.4.1, "Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint"
   - Section 6.7.5.1, "Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint"
   - Section 6.7.6.1, "Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint"
   - Section 6.7.7.1, "Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint"
   - Section 6.7.8, "Configuring a Non-buffered Destination for a Web Service"
   - Section 6.7.9, "Configuring the Acknowledgement Interval"
6. Click **Save**.

For more information, see "Web Service Reliable Messaging" in the *Oracle WebLogic Server Administration Console Help*.

6.7.1.2 Using WLST

Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

6.7.2 Configuring Reliable Messaging on the Web Service Endpoint

By default, Web service endpoints use the reliable messaging configuration defined for the server. You can override the reliable messaging configuration used by the Web service endpoint using the Administration Console, as follows:

> **Note:** Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

1. Invoke the Administration Console, as described in "Invoking the Administration Console" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
2. In the left navigation pane, select **Deployments**.
3. Click the name of the Web service in the Deployments table.
4. Select the **Configuration** tab, then the Port Components tab.
5. Click the name of the Web service endpoint in the Ports table.
6. Select the **Reliable Message** tab.
7. Click **Customize Reliable Message Configuration** and follow the instructions to save the deployment plan, if required.
8. Edit the reliable messaging properties, as described in the following sections:

- Section 6.7.4.1, "Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint"
- Section 6.7.5.1, "Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint"
- Section 6.7.6.1, "Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint"
- Section 6.7.7.1, "Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint"
- Section 6.7.8, "Configuring a Non-buffered Destination for a Web Service"
- Section 6.7.9, "Configuring the Acknowledgement Interval"

9. Click **Save**.

For more information, see "Configure Web Service Reliable Messaging" in the *Oracle WebLogic Server Administration Console Help*.

6.7.3 Configuring Reliable Messaging on Web Service Clients

For general information about configuring reliable messaging on Web service clients, see "Configuring Web Service Clients" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

For information about using the `weblogic.wsee.reliability2.api.WsrmClientInitFeature` when creating a Web services reliable messaging client, refer to the following sections:

- Section 6.7.4.2, "Configuring the Base Retransmission Interval on the Web Service Client"
- Section 6.7.5.2, "Configuring the Retransmission Exponential Backoff on the Web Service Client"
- Section 6.7.6.2, "Configuring Sequence Expiration on the Web Service Client"
- Section 6.7.7.2, "Configuring the Inactivity Timeout on the Web Service Client"

6.7.4 Configuring the Base Retransmission Interval

If the source endpoint does not receive an acknowledgement for a given message within the specified base retransmission interval, the source endpoint retransmits the message. The source endpoint can modify this retransmission interval at any point during the lifetime of the sequence of messages.

This interval can be used in conjunction with the retransmission exponential backoff, described in Section 6.7.5, "Configuring the Retransmission Exponential Backoff", to specify the algorithm that is used to adjust the retransmission interval.

The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `n`Y specifies the number of years, `n`M specifies the number of months, `n`D specifies the number of days, `T` is the date/time separator, `n`H specifies the number of hours, `n`M specifies the number of minutes, and `n`S specifies the number of seconds. This value defaults to `P0DT5S` (5 seconds).

The following sections describe how to configure the base retransmission interval:

- Section 6.7.4.1, "Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint"

- Section 6.7.4.2, "Configuring the Base Retransmission Interval on the Web Service Client"

6.7.4.1 Configuring the Base Retransmission Interval on WebLogic Server or the Web Service Endpoint

To configure the retransmission exponential backoff on WebLogic Server or the Web service endpoint level using the Administration Console, perform the following steps:

> **Note:** Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

1. Invoke the Administration Console and access the Web service reliable messaging pages at the server-level or Web service endpoint level, as described in the following sections, respectively:
   - Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server"
   - Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"
2. Set the **Base Retransmission Interval** value, as required.

6.7.4.2 Configuring the Base Retransmission Interval on the Web Service Client

> **Note:** For more information about configuring Web service clients, see "Configuring Web Service Clients" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 6–11 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring the interval of time that must pass before a message is retransmitted to the RM destination.

Table 6–11 Methods for Configuring the Base Retransmission Interval

Method	Description
`String getBaseRetransmissionInterval()`	Gets the base retransmission interval.
`void setBaseRetransmissionInterval(String interval)`	Sets the base retransmission interval.

In the following example, the base retransmission interval is set to 3 hours.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
```

```
    initFeature.setBaseRetransmissionInterval("P0DT3H");
    port = service.getMyReliableServicePort(initFeature);
...
```

The base retransmission interval configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
      <stub-property>
        <name>weblogic.wsee.wsrm.BaseRetransmissionInterval</name>
        <value>PT30S</value>
      </stub-property>
...
  </port-info>
</service-reference-description>
```

6.7.5 Configuring the Retransmission Exponential Backoff

The retransmission exponential backoff is used in conjunction with the base retransmission interval, described in Section 6.7.4, "Configuring the Base Retransmission Interval". If a destination endpoint does not acknowledge a sequence of messages for the time interval specified by the base retransmission interval, the exponential backoff algorithm is used for timing successive retransmissions by the source endpoint, should the message continue to go unacknowledged.

The exponential backoff algorithm specifies that successive retransmission intervals should increase exponentially, based on the base retransmission interval. For example, if the base retransmission interval is 2 seconds, and the exponential backoff element is set, successive retransmission intervals if messages continue to go unacknowledged are 2, 4, 8, 16, 32, and so on.

By default, this flag is disabled (false), indicating that the same retransmission interval is used in successive retries; the interval does not increase exponentially.

The following sections describe how to configure the retransmission exponential backoff:

- Section 6.7.5.1, "Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint"
- Section 6.7.5.2, "Configuring the Retransmission Exponential Backoff on the Web Service Client"

6.7.5.1 Configuring the Retransmission Exponential Backoff on WebLogic Server or Web Service Endpoint

To configure the retransmission exponential backoff on WebLogic Server or the Web service endpoint level using the Administration Console, perform the following steps:

> **Note:** Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

1. Invoke the Administration Console and access the Web service reliable messaging pages at the server-level or Web service endpoint level, as described in the following sections, respectively:

- Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server"
- Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"

2. Set the **Enable Retransmission Exponential Backoff** flag, as required.

6.7.5.2 Configuring the Retransmission Exponential Backoff on the Web Service Client

> **Note:** For more information about configuring Web service clients, see "Configuring Web Service Clients" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 6–12 defines the `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring whether the message retransmission interval will be adjusted using the retransmission exponential backoff algorithm.

Table 6–12 Methods for Configuring the Retransmission Exponential Backoff

Method	Description
`Boolean isRetransmissionExponentialBackoff()`	Indicates whether retransmission exponential backoff is enabled.
`void setBaseRetransmissionExponentialBackoff(boolean value)`	Specifies whether base retransmission exponential backoff is enabled. Valid values are `true` or `false`.

In the following example, the retransmission exponential backoff is enabled.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
    initFeature.setBaseRetransmissionInterval("P0DT3H");
    initFeature.setBaseRetransmissionExponentialBackoff(true);
    port = service.getMyReliableServicePort(initFeature);
...
```

The retransmission exponential backoff configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
            <name>weblogic.wsee.wsrm.RetransmissionExponentialBackoff</name>
            <value>true</value>
         </stub-property>
...
```

```
  </port-info>
</service-reference-description>
```

6.7.6 Configuring the Sequence Expiration

The sequence expiration specifies the expiration time for a sequence regardless of activity.

The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `n`Y specifies the number of years, `n`M specifies the number of months, `n`D specifies the number of days, `T` is the date/time separator, `n`H specifies the number of hours, `n`M specifies the number of minutes, and `n`S specifies the number of seconds. This value defaults to `P1D` (1 day).

The following sections describe how to configure the sequence expiration:

- Section 6.7.6.1, "Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint"
- Section 6.7.6.2, "Configuring Sequence Expiration on the Web Service Client"

6.7.6.1 Configuring the Sequence Expiration on WebLogic Server or Web Service Endpoint

To configure the sequence expiration on WebLogic Server or the Web service endpoint level using the Administration Console, perform the following steps:

> **Note:** Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

1. Invoke the Administration Console and access the Web service reliable messaging pages at the server-level or Web service endpoint level, as described in the following sections, respectively:
   - Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server"
   - Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"
2. Set the **Sequence Expiration** value, as required.

6.7.6.2 Configuring Sequence Expiration on the Web Service Client

> **Note:** For more information about configuring Web service clients, see "Configuring Web Service Clients" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 6–13 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for expiration time for a sequence regardless of activity.

Table 6–13 Methods for Configuring Sequence Expiration

Method	Description
`String getSequenceExpiration()`	Returns the sequence expiration currently configured.

Table 6–13 (Cont.) Methods for Configuring Sequence Expiration

Method	Description
`void setSequenceExpiration(String expiration)`	Expiration time for a sequence regardless of activity.

In the following example, the sequence expiration is set to 36 hours.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
    initFeature.setSequenceExpiration("P0DT36H");
    port = service.getMyReliableServicePort(initFeature);
...
```

The sequence expiration configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
            <name>weblogic.wsee.wsrm.SequenceExpiration</name>
            <value>PT10M</value>
         </stub-property>
...
  </port-info>
</service-reference-description>
```

6.7.7 Configuring Inactivity Timeout

If, during the inactivity timeout interval, an endpoint (the RM source or destination) has not received messages application or protocol messages, the endpoint may consider the RM sequence to have been terminated due to inactivity.

The value specified must be a positive value and conform to the XML schema duration lexical format, `P`*`n`*`Y`*`n`*`M`*`n`*`DT`*`n`*`H`*`n`*`M`*`n`*`S`, where *`n`*`Y` specifies the number of years, *`n`*`M` specifies the number of months, *`n`*`D` specifies the number of days, `T` is the date/time separator, *`n`*`H` specifies the number of hours, *`n`*`M` specifies the number of minutes, and *`n`*`S` specifies the number of seconds. This value defaults to `P0DT600S` (600 seconds).

The following sections describe how to configure the inactivity timeout:

- Section 6.7.7.1, "Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint"
- Section 6.7.7.2, "Configuring the Inactivity Timeout on the Web Service Client"

6.7.7.1 Configuring the Inactivity Timeout on WebLogic Server or Web Service Endpoint

To configure the inactivity timeout on WebLogic Server or the Web service endpoint level using the Administration Console, perform the following steps:

> **Note:** Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

1. Invoke the Administration Console and access the Web service reliable messaging pages at the server-level or Web service endpoint level, as described in the following sections, respectively:
   - Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server"
   - Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"
2. Set the **Inactivity Timeout** value, as required.

6.7.7.2 Configuring the Inactivity Timeout on the Web Service Client

> **Note:** For more information about configuring Web service clients, see "Configuring Web Service Clients" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 6–14 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring the inactivity timeout.

Table 6–14 Methods for Configuring Inactivity Timeout

Method	Description
`String getInactivityTimeout()`	Returns the inactivity timeout currently configured.
`void setInactivityTimeout(String timeout)`	Sets the inactivity timeout.

In the following example, the inactivity timeout interval is set to 1 hour.

```
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    WsrmClientInitFeature initFeature = new WsrmClientInitFeature(true);
    initFeature.setInactivityTimeout("P0DT1H");
    port = service.getMyReliableServicePort(initFeature);
...
```

The inactivity timeout configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
            <name>weblogic.wsee.wsrm.InactivityTimeout</name>
            <value>PT5M</value>
         </stub-property>
...
  </port-info>
</service-reference-description>
```

6.7.8 Configuring a Non-buffered Destination for a Web Service

You can control whether you want to disable message buffering on a particular destination server to control whether buffering is used when receiving messages. You can configure non-buffering on the destination server at the WebLogic Server or Web service endpoint level only, not at the Web service client level (buffering is enabled by default on a Web service client).

Note: If you configure a non-buffered destination, any Web service client that uses `@WebServiceRef` to define a reference to the configuration will receive responses without buffering them.

The non-buffered destination configuration appears in the `weblogic.xml` file as follows:

```
<service-reference-description>
...
   <port-info>
         <stub-property>
           <name>weblogic.wsee.wsrm.NonBufferedDestination</name>
           <value>true</value>
         </stub-property>
...
  </port-info>
</service-reference-description>
```

For more information about `@WebServiceRef`, see "Defining a Web Service Reference Using the @WebServiceRef Annotation" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

To configure the destination server to disable message buffering, on WebLogic Server or the Web service endpoint level using the Administration Console, perform the following steps:

Note: Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

---

1. Invoke the Administration Console and access the Web service reliable messaging pages at the server-level or Web service endpoint level, as described in the following sections, respectively:

- Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server"
- Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"

2. Set the **Non-buffered Destination** value, to configure the destination server, respectively, as required.

> **Note:** On the source server, message buffering should always be enabled. That is, the **Non-buffered Source** value should always be disabled.

6.7.9 Configuring the Acknowledgement Interval

The acknowledgement interval specifies the maximum interval during which the destination endpoint must transmit a stand-alone acknowledgement. You can configure the acknowledgement interval at the WebLogic Server or Web service endpoint level only, not at the Web service client level.

> **Note:** A Web service client that uses `@WebServiceRef` to define a reference to the Web service uses the acknowledgement interval value to control the amount of time that the client's response handling will wait until acknowledging responses that it receives. In other words, the client acts like an RM destination when receiving response messages.
>
> The non-buffered destination configuration appears in the `weblogic.xml` file as follows:
>
> ```
> <service-reference-description>
> ...
>    <port-info>
>          <stub-property>
>           <name>weblogic.wsee.wsrm.AcknowledgementInterval</name>
>           <value>PT5S</value>
>          </stub-property>
> ...
>   </port-info>
> </service-reference-description>
> ```
>
> For more information about `@WebServiceRef`, see "Defining a Web Service Reference Using the @WebServiceRef Annotation" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

A destination endpoint can send an acknowledgement on the return message immediately after it has received a message from a source endpoint, or it can send one separately as a stand-alone acknowledgement. If a return message is not available to send an acknowledgement, a destination endpoint may wait for up to the acknowledgement interval before sending a stand-alone acknowledgement. If there are no unacknowledged messages, the destination endpoint may choose not to send an acknowledgement.

The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `P0DT0.2S` (0.2 seconds).

To configure the acknowledgement interval, on WebLogic Server or the Web service endpoint level using the Administration Console, perform the following steps:

> **Note:** Alternatively, you can use WLST to configure reliable messaging. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

1. Invoke the Administration Console and access the Web service reliable messaging pages at the server-level or Web service endpoint level, as described in the following sections, respectively:
   - Section 6.7.1, "Configuring Reliable Messaging on WebLogic Server"
   - Section 6.7.2, "Configuring Reliable Messaging on the Web Service Endpoint"
2. Set the **Acknowledgement Interval** value, as required.

6.8 Implementing the Reliability Error Listener

To receive notifications related to reliability delivery failures in the event that a request cannot be delivered, you can implement the following `weblogic.wsee.reliability2.api.ReliabilityErrorListener` interface:

```
public interface ReliablityErrorListener {

   public void onReliabilityError(ReliabilityErrorContext context);
}
```

Table 6–15 defines that `weblogic.wsee.reliability2.api.WsrmClientInitFeature` methods for configuring the reliability error listener.

Table 6–15 Methods for Configuring the Reliability Error Listener

Method	Description
`ReliabilityErrorListener getReliabilityListener()`	Gets the reliability listener currently configured.
`void setErrorListener(ReliabilityErrorListener` *`errorListener`*`)`	Sets the reliability error listener.

The following provides an example of how to implement and use a reliability error listener in your Web service client. This example is excerpted from Example 5–1, "Reliable Web Service Client Best Practices Example".

```
import weblogic.wsee.reliability2.api.ReliabilityErrorListener;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    WsrmClientInitFeature rmFeature = new WsrmClientInitFeature();
    features.add(rmFeature);

    ReliabilityErrorListener listener = new ReliabilityErrorListener() {
      public void onReliabilityError(ReliabilityErrorContext context) {

        // At a *minimum* do this
```

```
        System.out.println("RM sequence failure: " +
                           context.getFaultSummaryMessage());
        _lastResponse = context.getFaultSummaryMessage();

        // And optionally do this...

        // The context parameter tells you whether a request or the entire
        // sequence has failed. If a sequence fails, you'll get a notification
        // for each undelivered request (if any) on the sequence.
        if (context.isRequestSpecific()) {
          // We have a single request failure (possibly as part of a larger
          // sequence failure).
          // We can get the original request back like this:
          String operationName = context.getOperationName();
          System.out.println("Failed to deliver request for operation '" +
                             operationName + "'. Fault summary: " +
                             context.getFaultSummaryMessage());
          if ("DoSomething".equals(operationName)) {
            try {
              String request = context.getRequest(JAXBContext.newInstance(),
                                                  String.class);
              System.out.println("Failed to deliver request for operation '" +
                                 operationName + "' with content: " +
                                 request);
              Map<String, Serializable> requestProps =
                context.getUserRequestContextProperties();
              if (requestProps != null) {
                // Fetch back any property you sent in
                // JAXWSProperties.PERSISTENT_CONTEXT when you sent the
                // request.
                String myProperty = (String)requestProps.get(MY_PROPERTY);
                System.out.println(myProperty + " failed!");
              }
            } catch (Exception e) {
              e.printStackTrace();
            }
          }
        } else {
          // The entire sequence has encountered an error.
          System.out.println("Entire sequence failed: " +
                             context.getFaultSummaryMessage());

        }
      }
    };

  rmFeature.setReliabilityErrorListener(listener);

  _features = features.toArray(new WebServiceFeature[features.size()]);

  BackendReliableService anotherPort =
    _service.getBackendReliableServicePort(_features);
...
```

6.9 Managing the Life Cycle of a Reliable Message Sequence

WebLogic Server provides a client API, `weblogic.wsee.reliability2.api.WsrmClient`, for use with the Web service

reliable messaging. Use this API to perform common life cycle tasks such as set configuration options, get the reliable sequence id, and terminate a reliable sequence.

An instance of the `WsrmClient` API can be accessed from the reliable Web service port using the `weblogic.wsee.reliability2.api.WsrmClientFactory` method, as follows:

```
package wsrm_jaxws.example;
import java.xml.ws.WebService;
import java.xml.ws.WebServiceRef;
import wsrm_jaxws.example.client_service.*;
import wsrm_jaxws.example.client_service.EchoResponse;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;
...
@WebService
public class ClientServiceImpl {
...
    @WebServiceRef(name="ReliableEchoService")
    private ReliableEchoService service;
    private ReliableEchoPortType port = null;
    port = service.getReliableEchoPort();
    WsrmClient wsrmClient = WsrmClientFactory.getWsrmClientFromPort(port);
...
```

The following sections describe how to manage the life cycle of a reliable message sequence using `WsrmClient`.

- Section 6.9.1, "Managing the Reliable Sequence"
- Section 6.9.2, "Managing the Client ID"
- Section 6.9.3, "Managing the Acknowledged Requests"
- Section 6.9.4, "Accessing Information About a Message"
- Section 6.9.5, "Identifying the Final Message in a Reliable Sequence"
- Section 6.9.6, "Closing the Reliable Sequence"
- Section 6.9.7, "Terminating the Reliable Sequence"
- Section 6.9.8, "Resetting a Client to Start a New Message Sequence"

For complete details on the Web service reliable messaging client API, see `weblogic.wsee.reliability2.api.WsrmClient` in *Oracle WebLogic Server API Reference*.

6.9.1 Managing the Reliable Sequence

To manage the reliable sequence, you can perform one or more of the following tasks.

- Get and set the reliable sequence ID, as described in Section 6.9.1.1, "Getting and Setting the Reliable Sequence ID".
- Access the state of the reliable sequence, for example, to determine if it is active or terminated, as described in Section 6.9.1.2, "Accessing the State of the Reliable Sequence".

6.9.1.1 Getting and Setting the Reliable Sequence ID

The sequence ID is used to identify a specific reliable sequence. You can get and set the sequence ID using the `weblogic.wsee.reliability2.api.WsrmClient.getSequenceID()` and `weblogic.wsee.reliability2.api.WsrmClient.setSequenceID()`

methods, respectively. If no messages have been sent when you issue the `getSequenceID()` method, the value returned is null.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
   _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   // Will be null
   String sequenceId = rmClient.getSequenceId();
   // Send first message
   anotherPort.doSomething("Bake a cake");
   // Will be non-null
   sequenceId = rmClient.getSequenceId();
```

During recovery from a server failure, you can set the reliable sequence on a newly created Web service port or dispatch instance after a client or server restart. Setting the sequence ID for a client instance is an advanced feature. Advanced clients may use `setSequenceId` to connect a client instance to a known RM sequence.

6.9.1.2 Accessing the State of the Reliable Sequence

To access the state of a sequence, use `weblogic.wsee.reliability2.api.WsrmClient.getSequenceState()`.This method returns an `java.lang.Enum` constant of the type `weblogic.wsee.reliability2.api.SequenceState`.

The following table defines valid values that may be returned for sequence state.

Table 6–16 Sequence State Values

Sequence State	Description
`CLOSED`	Reliable sequence is closed. **Note:** Closing a sequence should be considered a *last resort*, and only to prepare to close down a reliable messaging sequence for which you do not expect to receive the full range of requests. For more information, see Section 6.9.6, "Closing the Reliable Sequence"
`CLOSING`	Reliable sequence is in the process of being closed. **Note:** Closing a sequence should be considered a *last resort*, and only to prepare to close down a reliable messaging sequence for which you do not expect to receive the full range of requests. For more information, see Section 6.9.6, "Closing the Reliable Sequence"
`CREATED`	Reliable sequence has been created and the initial handshaking is complete.
`CREATING`	Reliable sequence is being created; the initial handshaking is in progress.

Table 6–16 (Cont.) Sequence State Values

Sequence State	Description
LAST_MESSAGE	**Deprecated**. WS-ReliableMessaging 1.0 only. The last message in the sequence has been received.
LAST_MESSAGE_PENDING	**Deprecated**. WS-ReliableMessaging 1.0 only. The last message in the sequence is pending.
NEW	Reliable sequence is in its initial state. Initial handshaking has not started.
TERMINATED	Reliable sequence is terminated. Under normal processing, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated. Though not recommended, you can force the termination of a reliable sequence, as described in Section 6.9.7, "Terminating the Reliable Sequence".
TERMINATING	Reliable sequence is in the process of being terminated. Under normal processing, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated. Though not recommended, you can force the termination of a reliable sequence, as described in Section 6.9.7, "Terminating the Reliable Sequence".

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
import weblogic.wsee.reliability2.api.SequenceState;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
        _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   SequenceState rmState = rmClient.getSequenceState();
   if (rmState == SequenceState.TERMINATED) {
         ... Do some work or log a message ...
   }
...
```

6.9.2 Managing the Client ID

The client ID identifies the Web service client. Each client has its own unique ID. The client ID can be used to access saved requests that may exist for a reliable sequence after a client or server restart.

The client ID is configured automatically by WebLogic Server. You can set the client ID to a custom value when creating the port using the `weblogic.wsee.jaxws.persistence.ClientIdentityFeature`. For more information, see "Managing Client Identity" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Reliable messaging uses the client ID to find any requests that were sent prior to a VM restart that were not sent before the VM exited. When you establish the first client instance using the prior client ID, reliable messaging uses the resources associated with that port to begin sending requests on behalf of the restored client ID.

You can get the client ID using the `weblogic.wsee.reliability2.api.WsrmClient.getID()` method.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
      _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   String clientId = rmClient.getId();
...
```

6.9.3 Managing the Acknowledged Requests

Use the `weblogic.wsee.reliability2.api.WsrmClient.ackRanges()` method to display the requests that have been acknowledged during the life cycle of a reliable message sequence. The `ackRanges()` method returns a set of `weblogic.wsee.reliability.MessageRange` objects.

After reviewing the range of requests that have been acknowledged, the client may choose to:

- Send an acknowledgement request to the RM destination using the `weblogic.wsee.reliability2.api.WsrmClient.requestAcknowledgement()` method.
- Close the sequence (see Section 6.9.6, "Closing the Reliable Sequence") and perform error handling to account for unacknowledged messages after a specific amount of time.

Note: Clients may call `getAckRanges()` repeatedly, to keep track of the reliable message sequence over time. However, you should take into account that there is a certain level of additional overhead associated each call.

6.9.4 Accessing Information About a Message

Use the `weblogic.wsee.reliability2.api.WsrmClient.getMessageInfo()` method to get information about a reliable message sent from the client based on the message number. This method accepts a long value representing the sequential message number of a request message sent from the client instance, and returns information about the message of type `weblogic.wsee.reliability2.sequence.SourceMessageInfo`. You can use the `WsrmClient.getMostRecentMessageNumber()` method to determine the maximum value of the message number value to pass to `getMessageInfo()`.

The returned `SourceMessageInfo` object should be treated as immutable, and only the get methods should be used.

The following table list the `SourceMessageInfo` methods that you can use to access specific details about the source message.

Table 6–17 Methods for SourceMessageInfo()

Method	Description
`getMessageID()`	Gets the message ID as a String value.
`getMessageNum()`	Gets the number of the message as a long value.
`getResponseMessageInfo()`	Returns a `weblogic.wsee.reliability2.sequence.DestinationMessageInfo` object representing the response that has been correlated to the request represented by the current `SourceMessageInfo()` object. Returns NULL if no response has been received for this request or if none is expected (for example, request was one way).
`isAck()`	Indicates whether the message has been acknowledged.

The following table lists the `DestinationMessageInfo` methods that you can use to access specific details about the destination message.

Table 6–18 Methods for DestinationMessageInfo()

Method	Description
`getMessageID()`	Gets the message ID as a String value.
`getMessageNum()`	Gets the number of the message as a long value.

The `getMessageInfo()` method can be used in conjunction with `weblogic.wsee.reliability2.api.WsrmClient.getMostRecentMessageNumber()` to obtain information about the most recently sent reliable message. This method returns a monotonically increasing long value, starting from 1. This method will return -1 in the following circumstances:

- If the reliable sequence ID has not been established (`getSequenceID()` returns null).
- The first reliable message has not been sent yet.
- The reliable sequence has been terminated.

6.9.5 Identifying the Final Message in a Reliable Sequence

Because WebLogic Server retains resources associated with the reliable sequence, it is recommended that you take steps to release these resources in a timely fashion. Under normal circumstances, a reliable sequence should be retained until all messages have been sent and acknowledged by the RM destination. To facilitate the timely and proper termination of a sequence, it is recommended that you identify the final message in a reliable message sequence. Doing so indicates you are done sending messages to the RM destination and that WebLogic Server can begin looking for the final acknowledgement before automatically terminating the reliable sequence. Indicate the final message using the `weblogic.wsee.reliability2.api.WsrmClient.setFinalMessage()` method.

When you identify a final message, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated, and all resources are released. Otherwise, the sequence is terminated automatically after the configured sequence expiration period is reached.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
   ...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
        _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   anotherPort.doSomething("One potato");
   anotherPort.doSomething("Two potato");
   anotherPort.doSomething("Three potato");
   // Indicate this next invoke marks the 'final' message for the sequence
   rmClient.setFinalMessage();
   anotherPort.doSomething("Four");
...
```

6.9.6 Closing the Reliable Sequence

Use the `weblogic.wsee.reliability2.api.WsrmClient.closeMessage()` to close a reliable messaging sequence.

> **Note:** This method is valid for WS-ReliableMessaging 1.1 only; it is not supported for WS-ReliableMessaging 1.0.

When a reliable messaging sequence is closed, no new messages will be accepted by the RM destination or sent by the RM source. A closed sequence is still tracked by the RM destination and continues to service acknowledgment requests against it. It allows the RM source to get a full and final accounting of the reliable messaging sequence before terminating it.

Note: Closing a sequence should be considered a *last resort*, and only to prepare to close down a reliable messaging sequence for which you do not expect to receive the full range of requests. For example, after reviewing the range of requests that have been acknowledged (see Section 6.9.3, "Managing the Acknowledged Requests"), the client may decide it necessary to close the sequence and perform error handling to account for unacknowledged messages after a specific amount of time.

Once a reliable messaging sequence is closed, it is up to the client to terminate the sequence; it will no longer be terminated automatically by the server after a configured timeout has been reached. See Section 6.9.7, "Terminating the Reliable Sequence".

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
```

```
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
      _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
   anotherPort.doSomething("One potato");
   anotherPort.doSomething("Two potato");
   // ... Wait some amount of time, and check for acks
   // ... using WsrmClient.getAckRanges() ...
   // ... If we don't find all of our acks ...
   rmClient.closeSequence();
   // ... Do some error recovery like telling our
   // ... client we couldn't deliver all requests ...
   rmClient.terminateSequence();
...
```

6.9.7 Terminating the Reliable Sequence

Although not recommended, you can terminate the reliable message sequence regardless of whether all messages have been acknowledged using the `weblogic.wsee.reliability2.api.WsrmClient.terminateSequence()` method.

> **Note:** It is recommended that, instead, you use the `setFinalMessage()` method to identify the final message in a reliable sequence. When you identify a final message, after all messages up to and including the final message are acknowledged, the reliable message sequence is terminated, and all resources are released. For more information, see Section 6.9.5, "Identifying the Final Message in a Reliable Sequence".

Terminating a sequence causes the RM source and RM destination to remove all state associated with that sequence. The client can no longer perform any action on a terminated sequence. When a sequence is terminated, any pending requests being delivered through server-side retry (SAF agents) for the sequence are rejected and sent as a notification on the `ReliablityErrorListener`.

For example:

```
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.api.WsrmClient;
...
   _service = new BackendReliableServiceService();
...
   features.add(... some features ...);
   _features = features.toArray(new WebServiceFeature[features.size()]);
...
   BackendReliableService anotherPort =
      _service.getBackendReliableServicePort(_features);
...
   WsrmClient rmClient = WsrmClientFactory.getWsrmClientFromPort(anotherPort);
...
```

```
   anotherPort.doSomething("One potato");
   anotherPort.doSomething("Two potato");
   // ... Wait some amount of time, and check for acks
   // ... using WsrmClient.getAckRanges() ...
   // ... If we don't find all of our acks ...
   rmClient.closeSequence();
   // ... Do some error recovery like telling our
   // ... client we couldn't deliver all requests ...
   rmClient.terminateSequence();
...
```

6.9.8 Resetting a Client to Start a New Message Sequence

Use the `weblogic.wsee.reliability2.api.WsrmClient.reset()` method to clear all `RequestContext` properties related to reliable messaging that do not need to be retained once the reliable sequence is closed. Typically, this method is called when you want to initiate another sequence of reliable messages from the same client.

For an example of using `reset()`, see Example B–1, "Example Client Wrapper Class for Batching Reliable Messages".

6.10 Monitoring Web Services Reliable Messaging

You can monitor reliable messaging sequences for a Web service or client using the Administration Console. For each reliable messaging sequence, runtime monitoring information is displayed, such as the sequence state, the source and destination servers, and so on. You can customize the information that is shown in the table by clicking **Customize this table**.

In particular, you can use the monitoring pages to determine:

- Whether or not you are cleaning up sequences in a timely fashion. If you view a large number of sequences in the monitoring tab, you may wish to review your client code to determine why.
- Whether an individual sequence has unacknowledged requests, or has not received expected responses.

To monitor reliable messaging sequences for a Web service, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the Web service is packaged. Expand the application by clicking the **+** node; the Web services in the application are listed under the **Web Services** category. Click on the name of the Web service and select **Monitoring> Ports> Reliable Messaging**.

To monitor reliable messaging sequences for a Web service client, click on the **Deployments** node in the left pane and, in the Deployments table that appears in the right pane, locate the Enterprise application in which the Web service client is packaged. Expand the application by clicking the **+** node and click on the application module within which the Web service client is located. Click the **Monitoring** tab, then click the **Web Service Clients** tab. Then click **Monitoring> Servers> Reliable Messaging**.

6.11 Grouping Messages into Business Units of Work (Batching)

Often, the messages flowing between a Web service client and service are part of a single business transaction or *unit of work*. An example might be a travel agency

reservation process that requires messages between the agency, airline, hotel, and rental car company. All of the messages flowing between any two endpoints could be considered a business unit of work.

Reliable messaging is tailored to handling messages related to a unit of work by grouping them into an RM sequence. The entire unit of work (or sequence) is treated as a whole, and error recovery, and so on can be applied to the entire sequence (see the `IncompleteSequenceBehavior` element description in the WS-ReliableMessaging 1.2 specification (February 2009) at `http://docs.oasis-open.org/ws-rx/wsrm/200702`). For example, an RM sequence can be configured to discard requests that occur after a gap in the sequence, or to discard the entire sequence of requests if any request is missing from the sequence.

You can indicate that a message is part of a business unit of work by creating a new client instance before sending the first message in the unit, and by disposing of the client instance after the last message in the unit. Alternatively, you can use the `WsrmClient` API (obtained by passing a client instance to the `WsrmClientFactory.getWsrmClientFromPort()` method) to identify the *final* request in a sequence is about to be sent. This is done by calling `WsrmClient.setFinalMessage()` just before performing the invoke on the client instance, as described in Section 6.9.5, "Identifying the Final Message in a Reliable Sequence."

There is some significant overhead associated with the RM protocol. In particular, creating and terminating a sequence involves a round-trip message exchange with the service (RM destination). This means that *four* messages must go across the wire to establish and then terminate an RM sequence. For this reason, it is to your advantage to send the requests within a single business unit of work on a single RM sequence. This allows you to amortize the cost of the RM protocol overhead over a number of business messages.

In some cases, the client instance being used to talk to the reliable service runs in an environment where there is no intrinsic notion of the business unit of work to which the messages belong. An example of this is an intermediary such as a message broker. In this case, the broker is often aware only of the message itself, and not the context in which the message is being sent. The broker may not do anything to demarcate the start and end of a business unit of work (or sequence); as a result, when using reliable messaging to send requests, the broker will incur the RM sequence creation and termination protocol overhead for *every* message it sends. This can result in a serious negative performance impact.

In cases where no intrinsic business unit of work is known for a message, you can choose to arbitrarily group (or batch) messages into an artificially created unit of work (called a *batch*). Batching of reliable messages can overcome the performance impact described above and can be used to tune and optimize network usage and throughput between a reliable messaging client and service. Testing has shown that batching otherwise unrelated requests into even small batches (say 10 requests) can as much as triple the throughput between the client and service when using reliable messaging (when sending small messages).

Note: Oracle does not recommend batching requests that already have an association with a business unit of work. This is because error recovery can become complicated when RM sequence boundaries and unit of work boundaries do not match. For example, when you add a `ReliabilityErrorListener` to your client instance (via `WsrmClientInitFeature`), as described in Section 6.8, "Implementing the Reliability Error Listener," this listener can be used to perform error recovery for single requests in a sequence or whole-sequence failures. When batching requests, this error recovery logic would need to store some information about each request in order to properly handle the failure of a request. A client that does not employ batching will likely have more context about the request given the business unit of work it belongs to.

The following code excerpt shows an example class called `BatchingRmClientWrapper` that can be used to make batching of RM requests simple and effective. This class batches requests into groups of a specified number of requests. It allows you to create a dynamic proxy that takes the place of your regular client instance. When you make invocations on the client instance, the batching wrapper seamlessly groups the outgoing requests into batches, and assigns each batch its own RM sequence. The batching wrapper also takes a duration specification that indicates the maximum lifetime of any given batch. This allows incomplete batches to be completed in a timely fashion even if there are not enough outgoing requests to completely fill a batch. If the batch has existed for the maximum lifetime specified, it will be closed as if the last message in the batch had been sent.

An example of the client wrapper class that can be used for batching reliable messaging is provided in Appendix B, "Example Client Wrapper Class for Batching Reliable Messages". You can use this class as-is in your own application code, if desired.

Example 6–2 Example of Grouping Messages into Units of Work (Batching)

```
import java.io.IOException;
import java.util.*;
import java.util.*;

import javax.servlet.*;
import javax.xml.ws.*;

import weblogic.jws.jaxws.client.ClientIdentityFeature;
import weblogic.jws.jaxws.client.async.AsyncClientHandlerFeature;
import weblogic.jws.jaxws.client.async.AsyncClientTransportFeature;
import weblogic.wsee.reliability2.api.ReliabilityErrorContext;
import weblogic.wsee.reliability2.api.ReliabilityErrorListener;
import weblogic.wsee.reliability2.api.WsrmClientInitFeature;

/**
 * Example client for invoking a reliable Web service and 'batching' requests
 * artificially into a sequence. A wrapper class called
 * BatchingRmClientWrapper is called to begin and end RM sequences for each batch of
 * requests. This avoids per-message RM sequence handshaking
 * and termination overhead (delivering better performance).
 */
public class BestPracticeAsyncRmBatchingClient
  extends GenericServlet {
```

```
private BackendReliableServiceService _service;
private BackendReliableService _singletonPort;
private BackendReliableService _batchingPort;

private static int _requestCount;
private static String _lastResponse;

@Override
public void init()
  throws ServletException {

  _requestCount = 0;
  _lastResponse = null;

  // Only create the Web service object once as it is expensive to create repeatedly.
  if (_service == null) {
    _service = new BackendReliableServiceService();
  }

  // Best Practice: Use a stored list of features, per client ID, to create client instances.
  // Define all features for the Web service port, per client ID, so that they are
  // consistent each time the port is called. For example:
  // _service.getBackendServicePort(_features);

  List<WebServiceFeature> features = new ArrayList<WebServiceFeature>();

  // Best Practice: Explicitly define the client ID.
  ClientIdentityFeature clientIdFeature =
    new ClientIdentityFeature("MyBackendServiceAsyncRmBatchingClient");
  features.add(clientIdFeature);

  // Best Practice: Always implement a reliability error listener.
  // Include this feature in your reusable feature list. This enables you to determine
  // a reason for failure, for example, RM cannot deliver a request or the RM sequence fails in
  // some way (for example, client credentials refused at service).
  WsrmClientInitFeature rmFeature = new WsrmClientInitFeature();
  features.add(rmFeature);
  rmFeature.setErrorListener(new ReliabilityErrorListener() {
    public void onReliabilityError(ReliabilityErrorContext context) {
      // At a *minimum* do this
      System.out.println("RM sequence failure: " +
                         context.getFaultSummaryMessage());
      _lastResponse = context.getFaultSummaryMessage();
    }
  });

  // Asynchronous endpoint
  AsyncClientTransportFeature asyncFeature =
    new AsyncClientTransportFeature(getServletContext());
  features.add(asyncFeature);

  // Best Practice: Define a port-based asynchronous callback handler,
  // AsyncClientHandlerFeature, for asynchronous and dispatch callback handling.
  BackendReliableServiceAsyncHandler handler =
    new BackendReliableServiceAsyncHandler() {
      public void onDoSomethingResponse(Response<DoSomethingResponse> res) {
        // ... Handle Response ...
        try {
          DoSomethingResponse response = res.get();
          _lastResponse = response.getReturn();
```

```
          System.out.println("Got reliable/async/batched response: " + _lastResponse);
        } catch (Exception e) {
          _lastResponse = e.toString();
          e.printStackTrace();
        }
      }
    };
  AsyncClientHandlerFeature handlerFeature =
    new AsyncClientHandlerFeature(handler);
  features.add(handlerFeature);

  // Set the features used when creating clients with
  // this client ID "MyBackendServiceAsyncRmBatchingClient"

  WebServiceFeature[] featuresArray =
    features.toArray(new WebServiceFeature[features.size()]);

  // Best Practice: Define a singleton port instance and initialize it when
  // the client container initializes (upon deployment).
  // The singleton port will be available for the life of the servlet.
  // Creation of the singleton port triggers the asynchronous response endpoint to be published
  // and it will remain published until our container (Web application) is undeployed.
  // Note, we will get a call to destroy() before this.
  _singletonPort = _service.getBackendReliableServicePort(featuresArray);

  // Create a wrapper class to 'batch' messages onto RM sequences so
  // a client with no concept of which messages are related as a unit can still achieve
  // good performance from RM. The class will send a given number of requests on
  // the same sequence, and then terminate that sequence before starting
  // another to carry further requests. A batch has both a max size and
  // lifetime so no sequence is left open for too long.
  // The example batches 10 messages or executes for 20 seconds, whichever comes
  // first. Assuming there were 15 total requests to send, the class would start and complete
  // one full batch of 10 requests, then send the next batch of five requests.
  // Once the batch of five requests has been open for 20 seconds, it will be closed and the
  // associated sequence terminated (even though 10 requests were not sent to fill the batch).
  BackendReliableService batchingPort =
    _service.getBackendReliableServicePort(featuresArray);
  BatchingRmClientWrapper<BackendReliableService> batchingSeq
    = new BatchingRmClientWrapper<BackendReliableService>(batchingPort,
                                                BackendReliableService.class,
                                                10, "PT20S",
                                                System.out);
  _batchingPort = batchingSeq.createProxy();
}

@Override
public void service(ServletRequest req, ServletResponse res)
  throws ServletException, IOException {

  // TODO: ... Read the servlet request ...

  // For this simple example, echo the _lastResponse captured from
  // an asynchronous DoSomethingResponse response message.

  if (_lastResponse != null) {
    res.getWriter().write(_lastResponse);
    System.out.println("Servlet returning _lastResponse value: " + _lastResponse);
    _lastResponse = null; // Clear the response so we can get another
    return;
```

```
    }

    // Synchronize on _batchingPort since it is a class-level variable and it might
    // be in this method on multiple threads from the servlet engine.

    synchronized(_batchingPort) {

      // Use the 'batching' port to send the requests instead of creating a
      // new request each time.
      BackendReliableService port = _batchingPort;

      // Set the endpoint address for BackendService.
      ((BindingProvider)port).getRequestContext().
        put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
            "http://localhost:7001/BestPracticeReliableService/BackendReliableService");

    // Make the invocation. Our asynchronous handler implementation (set
    // into the AsyncClientHandlerFeature above) receives the response.
      String request = "Protected and serve " + (++_requestCount);
      System.out.println("Invoking DoSomething reliably/async/batched with request: " +
                         request);
      port.doSomethingAsync(request);
    }

    // Return a canned string indicating the response was not received
    // synchronously. Client needs to invoke the servlet again to get
    // the response.
    res.getWriter().write("Waiting for response...");
  }

  @Override
  public void destroy() {

    try {
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the singleton port created during initialization. Note, the asynchronous
      // response endpoint generated by creating _singletonPort *remains*
      // published until our container (Web application) is undeployed.
      ((java.io.Closeable)_singletonPort).close();
      // Best Practice: Explicitly close client instances when processing is complete.
      // Close the batching port created during initialization. Note, this will close
      // the underlying client instance used to create the batching port.
      ((java.io.Closeable)_batchingPort).close();

      // Upon return, the Web application is undeployed, and the asynchronous
      // response endpoint is stopped (unpublished). At this point,
      // the client ID used for _singletonPort will be unregistered and will no longer be
      // visible from the Administration Console and WLST.
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

6.12 Client Considerations When Redeploying a Reliable Web Service

WebLogic Server supports production redeployment, which means that you can deploy a new version of an updated reliable WebLogic Web service alongside an older version of the same Web service.

WebLogic Server automatically manages client connections so that only *new* client requests are directed to the new version. Clients already connected to the Web service during the redeployment continue to use the older version of the service until they complete their work, at which point WebLogic Server automatically retires the older Web service. If the client is connected to a reliable Web service, its work is considered complete when the existing reliable message sequence is explicitly ended by the client or as a result of a timeout.

For additional information about production redeployment and Web service clients, see "Client Considerations When Redeploying a Web service" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

6.13 Interoperability with WebLogic Web Service Reliable Messaging

The WebLogic Web services reliable messaging implementation will interoperate with the Web service reliable messaging implementations provided by the following third-party vendor Web services: IBM and Microsoft .NET. For best practices when interoperating with Microsoft .NET, see "Interoperability with Microsoft WCF/.NET" in *Introducing WebLogic Web Services for Oracle WebLogic Server*.

To enhance interoperability with Oracle SOA services that use Web services reliable messaging, please consider the following interoperability guidelines:

- Do no use Make Connection for asynchronous transport, as described in Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)." Reliable SOA services do not support Make Connection.
- Do no use WS-SecureConversation to secure reliable Web services. SOA services do not support the use of Web services reliable messaging and WS-SecureConversation together.
- For reliable WebLogic Web service clients that are accessing reliable SOA services:
  - Use synchronous (anonymous WS-Addressing ReplyTo EPR) request-reply or one-way MEP (Message exchange pattern).
  - Do **not** use asynchronous (non-anonymous WS-Addressing ReplyTo EPR) request-reply MEP.
- For reliable SOA clients that are accessing reliable WebLogic Web services, use one of the following:
  - Synchronous (anonymous WS-Addressing ReplyTo EPR) request-reply or one-way MEP.
  - Asynchronous (non-anonymous WS-Addressing ReplyTo EPR) request-reply MEP.

7

Managing Web Service Persistence

This chapter describes how to manage persistence for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

7.1 Overview of Web Service Persistence

WebLogic Server provides a default Web service persistence configuration that provides a built-in, high-performance storage solution for Web services. Web service persistence is used by the following advanced features to support long running requests and to survive server restarts:

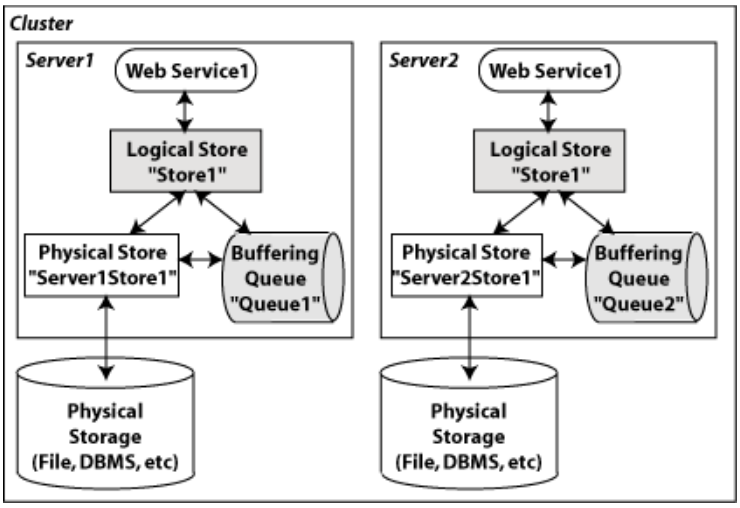
- Asynchronous Web service invocation using asynchronous client transport or Make Connection
- Web services reliable messaging
- Message buffering
- Security using WS-SecureConversation

Specifically, Web service persistence is used to save the following types of information:

- Client identity and properties
- SOAP message, including its headers and body
- Context properties required for processing the message at the Web service or client (for both asynchronous and synchronous messages)

The following figure illustrates an example Web service persistence configuration.

Figure 7–1 Example Web Service Persistence Configuration



The following table describes the components of Web service persistence, shown in the previous figure.

Table 7–1 Components of the Web Service Persistence

Component	Description
Logical Store	Provides the configuration requirements and connects the Web service to the physical store and buffering queue.
Physical store	Handles the I/O operations to save and retrieve data from the physical storage (such as file, DBMS, and so on). The physical store can be a WebLogic Server persistent store, as configured using the WebLogic Server Administration Console or WLST, or in-memory store. **Note**: When using a WebLogic Server persistent store as the physical store for a logical store, the names of the request and response buffering queues are taken from the logical store configuration and not the buffering configuration.
Buffering queue	Stores buffered requests and responses for the Web service.

When configuring Web service persistence, you associate:

- A logical store with a buffering queue.
- A buffering queue that is associated with a physical store via JMS configuration.

The association between the logical store and buffering queue is used to infer the association between the logical store and physical store. The default logical store is named `WseeStore` and is created automatically when a domain is created using the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`). By default, the physical store that is configured for the server is associated with the buffering queue. This strategy ensures that the same physical store is used for all Web service persistence and buffering. Using a single physical store ensures a more efficient, single-phase XA transaction and facilitates migration.

You can configure one or more logical stores for use within your application environment. In Table 7–1, the servers `Server1` and `Server2` use the same logical store. This configuration allows applications that are running in a cluster to be configured globally to access a single store name. As described later in Section 7.3, "Configuring Web Service Persistence", you can configure Web service persistence at

various levels for fine-grained management. Best practices are provided in Section 7.2, "Roadmap for Configuring Web Service Persistence."

7.2 Roadmap for Configuring Web Service Persistence

Table 7–2 provides best practices for configuring Web service persistence to support Web service reliable messaging.

Table 7–2 Roadmap for Configuring Web Service Persistence

Best Practice	Description
Define a logical store for each administrative unit (for example, business unit, department, and so on).	By defining separate logical stores, you can better manage the service-level agreements for each administrative unit. For more information, see Section 7.3.1, "Configuring the Logical Store."
Use the correct logical store for each client or service related to the administrative unit.	You can configure the logical store at the WebLogic Server, Web service, or Web service client level. For more information, see Section 7.3, "Configuring Web Service Persistence."
Define separate physical stores and buffering queues for each logical store.	For more information, see Section 7–1, "Example Web Service Persistence Configuration."

The best practices defined in Table 7–2 facilitates maintenance, and failure recovery and resource migration.

For example, assume Company X is developing Web services for several departments, including manufacturing, accounts payable, accounts receivable. Following best practices, Company X defines a minimum of three logical stores, one for each department.

Furthermore, assume that the manufacturing department has a service-level agreement with the IT department that specifies that it can tolerate system outages that are no longer than a few minutes in duration. The accounts payable and receivable departments, on the other hand, have a more relaxed service-level agreement, tolerating system outages up to one hour in duration. If the systems that host Web services and clients for the manufacturing department become unavailable, the IT department is responsible for ensuring that any resources required by those Web services and clients are migrated to new active servers within minutes of the failure. Because separate logical stores were defined, the IT department can migrate the file store, JMS servers, and so on, associated with the manufacturing department logical store independently of the resources required for accounts payables and receivables.

7.3 Configuring Web Service Persistence

The following table summarizes the information that you can configure for each of the Web service persistence components.

Table 7–3 Summary of the Web Service Persistence Component Configuration

Component	Summary of Configuration Requirements
Logical Store	You configure the following information for each logical store: ■ Name of the logical store. ■ Maximum lifetime of an object in the store. ■ The cleaner thread that removes stale objects from the store. For more information, see Section 7.5, "Cleaning Up Web Service Persistence". ■ Accessibility from other servers in a network. ■ Request and response buffering queues. The request buffering queue is used to infer the physical store by association.
Physical store	You configure the following information for the physical store: ■ Name of the physical store. ■ Type and performance parameters. ■ Location of the store. **Note**: You configure the physical store or buffering queue, but not both. If the buffering queue is configured, then the physical store information is inferred.
Buffering queue	You configure the following information for the buffering queue: ■ Request and response queue details ■ Retry counts and delays

You can configure Web service persistence at the levels defined in the following table.

Table 7–4 Configuring Web Service Persistence

Level	Description
WebLogic Server	The Web service persistence configured at the server level defines the default configuration for all Web services and clients running on that server. To configure Web service persistence for WebLogic Server, use one of the following methods: ■ When creating or extending a domain using Configuration Wizard, you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`) to configure automatically the resources required to support Web services persistence. Although use of this extension template is not required, it makes the configuration of the required resources much easier. ■ Configure the resources required for Web service persistence using the Oracle WebLogic Administration Console or WLST. For more information, see: - Administration Console: "Configure Web service persistence" in *Oracle WebLogic Server Administration Console Help* - WLST: "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool* For more information, see "Configuring Your Domain for Advanced Web Services Features" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
Web service endpoint	Configure the default logical store used by the Web service endpoint, as described in Section 7.3.2, "Configuring Web Service Persistence for a Web Service Endpoint."
Web service client	Configure the default logical store used by the Web service client, as described in Section 7.3.3, "Configuring Web Service Persistence for Web Service Clients."

The following sections provide more information about configuring Web service persistence:

- Section 7.3.1, "Configuring the Logical Store"

- Section 7.3.2, "Configuring Web Service Persistence for a Web Service Endpoint"
- Section 7.3.3, "Configuring Web Service Persistence for Web Service Clients"

7.3.1 Configuring the Logical Store

You can configure one or more logical stores for use within your application environment, and identify the logical store that is used as the default.

The default logical store, `WseeStore`, is generated automatically when you create or extend a domain using the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`), as described in "Configuring Your Domain for Advanced Web Services Features" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

You can configure the logical store using the Administration Console, see "Configure Web service persistence" in *Oracle WebLogic Server Administration Console Help*. Alternatively, you can use WLST to configure the resources. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

The following table summarizes the properties that you define for the logical store.

Table 7–5 Configuration Properties for the Logical Store

Property	Description
Logical Store Name	Name of the logical store. The name must begin with an alphabetical character and can contain alphabetical characters, spaces, dashes, underscores, and numbers only. This field defaults to `LogicalStore_n`. This field is required. If you create or extend a single server domain using the Web service extension template, a logical store named `WseeStore` is created by default.
Default Logical Store	Flag that specifies whether the logical store is used, by default, to persist state of all Web services on the server. Only one logical store can be set as the default. If you enable this flag on the current logical store, the flag is disabled on the current default store.
Persistence strategy	Persistence strategy. Select one of the following values from the drop-down menu. ■ `Local Access Only`—Accessible to the local server only. ■ `In Memory`—Accessible by the local VM only. In this case, the buffering queue and physical store configuration information is ignored.
Request Buffering Queue JNDI Name	JNDI name for the request buffering queue. The request buffering queue is used to infer the physical store by association. If this property is not set, then the default physical store that is configured for the server is used. **Note**: You configure the physical store or buffering queue, but not both. If the buffering queue is configured, then the physical store information is inferred. It is recommended that the same physical storage resource be used for both persistent state and message buffering to allow for a more efficient, single-phase XA transaction and facilitate service migration. By setting this value, you ensure that the buffering queue and physical store reference the same physical storage resource. If you create or extend a domain using the Web service extension template, a buffering queue named `weblogic.wsee.RequestBufferedRequestQueue` is created by default. **Note**: This property is ignored if **Persistence strategy** is set to `In Memory`.

Table 7–5 (Cont.) Configuration Properties for the Logical Store

Property	Description
Response Buffering Queue JNDI Name	JNDI name for the response buffering queue. If this property is not set, then the request queue is used, as defined by the **Request Buffering Queue JNDI Name** property. If you create or extend a domain using the Web service extension template, a buffering queue named `weblogic.wsee.RequestBufferedRequestErrorQueue` is created by default. **Note**: This property is ignored if **Persistence strategy** is set to `In Memory`.
Cleaner Interval	Interval at which the logical store will be cleaned. For more information, see Section 7.5, "Cleaning Up Web Service Persistence" The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `PT10M` (10 minutes). **Note**: This field is available when editing the logical store only. When creating the logical store, the field is set to the default, `PT10M` (10 minutes).
Default Maximum Object Lifetime	Default value used as the maximum lifetime of an object. This value can be overridden by the individual objects saved to the logical store. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where *n*`Y` specifies the number of years, *n*`M` specifies the number of months, *n*`D` specifies the number of days, `T` is the date/time separator, *n*`H` specifies the number of hours, *n*`M` specifies the number of minutes, and *n*`S` specifies the number of seconds. This value defaults to `P1D` (one day). **Note**: This field is available when editing the logical store only. When creating the logical store, the field is set to the default, `P1D` (one day).

7.3.2 Configuring Web Service Persistence for a Web Service Endpoint

By default, Web service endpoints use the Web service persistent store defined for the server. You can override the logical store used by the Web service endpoint using the Administration Console. For more information, see "Configure Web service persistence" in *Oracle WebLogic Server Administration Console Help*.

7.3.3 Configuring Web Service Persistence for Web Service Clients

For information about configuring persistence for Web service clients, see "Configuring Web Service Clients" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

7.4 Using Web Service Persistence in a Cluster

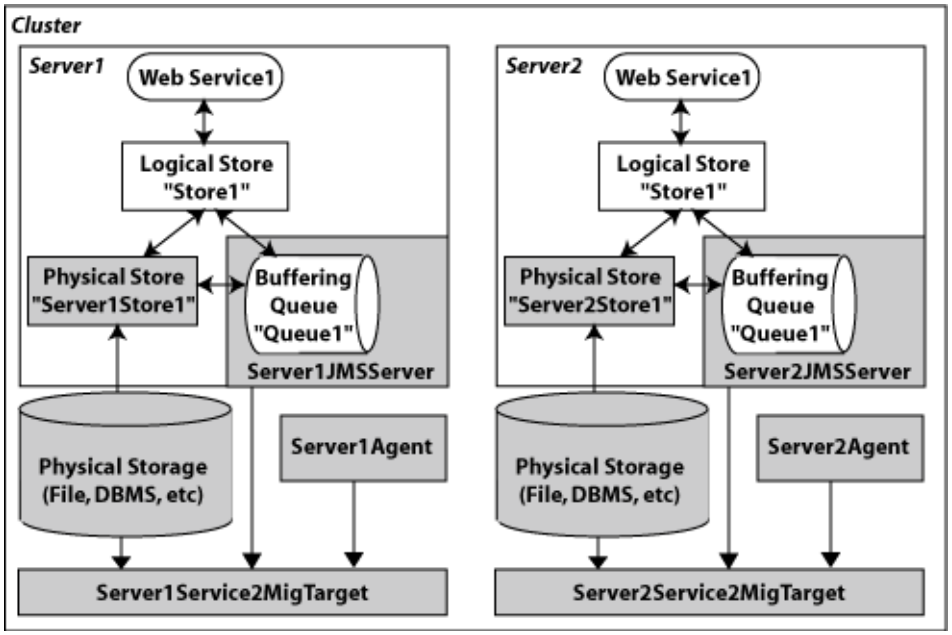
The following provides some considerations for using Web services persistence in a cluster:

- If you create or extend a clustered domain using the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`), the resources required to support Web services persistence in a cluster are automatically created. For more information, see "Configuring Your Domain for Advanced Web Services Features" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*

- To facilitate service migration, it is recommended that the same physical storage resource be used for both persistent state and message buffering. To ensure that the buffering queue and physical store reference the same physical storage resource, you configure the **Request Buffering Queue JNDI Name** property of the logical store, as described in Section 7.3.1, "Configuring the Logical Store".
- It is recommended that the buffering queues be defined as JMS uniform distributed destinations (UDDs). JMS defines a member queue for the UDD on each JMS Server that you identify. Because a logical store is associated with a physical store through the defined buffering queue, during service migration, this allows a logical store to use the new physical stores seamlessly for the member queues that migrate onto the new server.
- It is recommended that you target the JMS Server, store-and-forward (SAF) service agent, and physical store (file store) resources to migrateable targets. For more information, see "Resources Required by Advanced Web Service Features" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

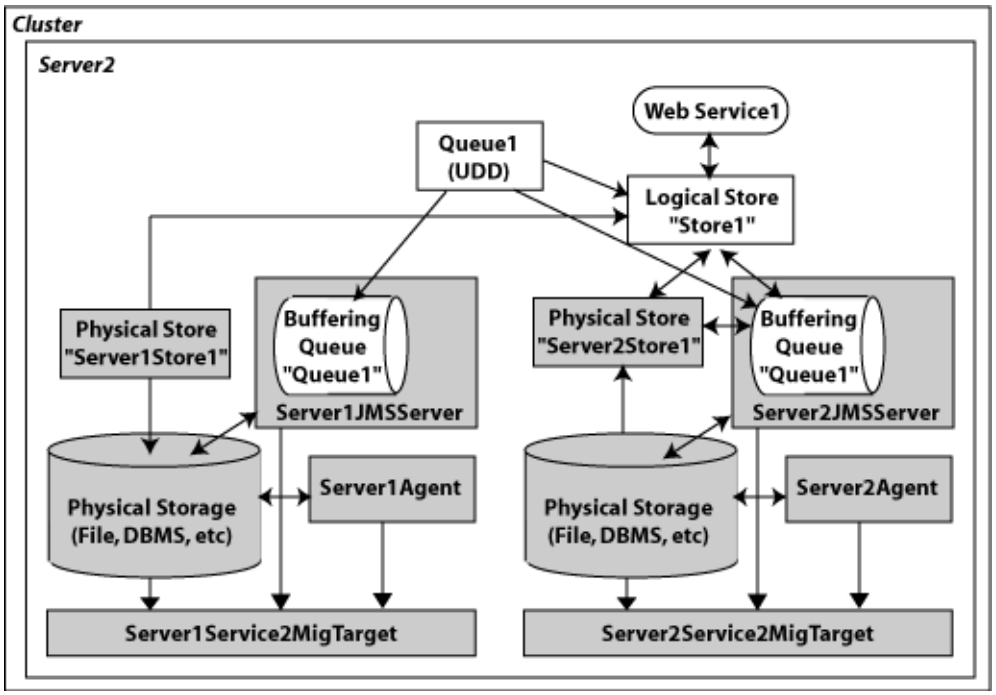
For example, consider the two-node cluster configuration shown in Figure 7–2. The domain resources are configured and targeted using the guidelines provided above.

Figure 7–2 Example of a Two-Node Cluster Configuration (Before Migration)



The following figure shows how the resources on Server1 can be easily migrated to Server2 in the event Server1 fails.

Figure 7–3 Example of a Two-Node Cluster Configuration (After Migration)



7.5 Cleaning Up Web Service Persistence

The persisted information is cleaned up periodically to remove expired or stale objects. Typically, an object is associated with a specific expiration time or a maximum lifetime. In addition, a *stale* object may represent a request for which no response was received or a reliable messaging sequence that was not explicitly terminated.

You configure the interval of time at which Web service persistence will be cleaned by setting the **Cleaner Interval** configuration property on the logical store. For more information about setting this property, see Section 7.3.1, "Configuring the Logical Store".

8

Configuring Message Buffering for Web Services

This chapter describes how to configure message buffering for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

- Section 8.1, "Overview of Message Buffering"
- Section 8.2, "Configuring Messaging Buffering"

8.1 Overview of Message Buffering

When an operation on a buffered Web service is invoked, the message representing that invocation is stored in a JMS queue. WebLogic Server processes this buffered message asynchronously. If WebLogic Server goes down while the message is still in the queue, it will be processed as soon as WebLogic Server is restarted.

WebLogic Server then processes the request message on a separate thread obtained from a pre-configured and managed pool of threads. This allows WebLogic Server to absorb spikes in client load, and continue to process the requests in an orderly fashion over a period of time. Message buffering is a powerful tool to avoid denial of service attacks and general overload conditions on the server.

To assist you in determining whether to configure message buffering on the Web service, it is recommended that you review Section 6.1.4.4, "Failure Scenarios with Non-buffered Reliable Web Services."

8.2 Configuring Messaging Buffering

You can configure message buffering for Web services at the WebLogic Server or Web service endpoint levels. The message buffering configured at the server level defines the default message buffering defined for all Web services and clients running on that server, unless explicitly overridden at the Web service endpoint level.

For detailed steps to configure message buffering for Web services at the WebLogic Server or Web service endpoint level using the WebLogic Server Administration Console, see "Configure message buffering for Web services" in *Oracle WebLogic Server Administration Console Help*.

When you configure message buffering at the Web service endpoint level, select **Customize Buffering Configuration** to indicate that you want to customize the buffering configuration defined in the Web service descriptor or deployment plan at the Web service endpoint level. If not checked, the buffering configuration specified at the WebLogic Server level is used.

Alternatively, you can use WLST to configure message buffering. For information about using WLST to extend the domain, see "Configuring Existing Domains" in *Oracle WebLogic Scripting Tool*.

The following sections describe message buffering configuration properties:

- Section 8.2.1, "Configuring the Request Queue"
- Section 8.2.2, "Configuring the Response Queue"
- Section 8.2.3, "Configuring Message Retry Count and Delay"

8.2.1 Configuring the Request Queue

The following table summarizes the properties used to configure the request queue.

Table 8–1 Configuring the Request Queue

Property	Description
Request Queue Enabled	Flag that specifies whether the request queue is enabled. By default, the request queue is disabled. The request queue name is defined by the logical store enabled at this level. When using a WebLogic Server persistent store as the physical store for a logical store, the names of the request and response buffering queues are taken from the logical store configuration and not the buffering configuration.
Request Queue Connection Factory JNDI Name	JNDI name of the connection factory to use for request message buffering. This value defaults to the default JMS connection factory defined by the server.
Request Queue Transaction Enabled	Flag that specifies whether transactions should be used when storing and retrieving messages from the request buffering queue. This flag defaults to false.

8.2.2 Configuring the Response Queue

The following table summarizes the properties used to configure the response queue.

Table 8–2 Configuring the Response Queue

Property	Description
Response Queue Enabled	Flag that specifies whether the response queue is enabled. By default, the response queue is disabled. The response queue name is defined by the logical store enabled at this level. When using a WebLogic Server persistent store as the physical store for a logical store, the names of the request and response buffering queues are taken from the logical store configuration and not the buffering configuration.
Response Queue Connection Factory JNDI Name	JNDI name of the connection factory to use for response message buffering. This value defaults to the default JMS connection factory defined by the server.
Response Queue Transaction Enabled	Flag that specifies whether transactions should be used when storing and retrieving messages from the response buffering queue. This flag defaults to false.

8.2.3 Configuring Message Retry Count and Delay

The following table summarizes the properties used to configure the message retry count and delay.

Table 8–3 Configuring Message Retry Count and Delay

Property	Description
Retry Count	Number of times that the JMS queue on the invoked WebLogic Server instance attempts to deliver the message to the Web service implementation until the operation is successfully invoked. This value defaults to 3.
Retry Delay	Amount of time between retries of a buffered request and response. Note, this value is only applicable when **RetryCount** is greater than 0. The value specified must be a positive value and conform to the XML schema duration lexical format, `PnYnMnDTnHnMnS`, where `n`Y specifies the number of years, `n`M specifies the number of months, `n`D specifies the number of days, `T` is the date/time separator, `n`H specifies the number of hours, `n`M specifies the number of minutes, and `n`S specifies the number of seconds. This value defaults to `P0DT30S` (30 seconds).

9

Managing Web Services in a Cluster

This chapter describes how to manage WebLogic Web services in a cluster.

This chapter includes the following sections:

- Section 9.1, "Overview of Web Services Cluster Routing"
- Section 9.2, "Cluster Routing Scenarios"
- Section 9.3, "How Web Service Cluster Routing Works"
- Section 9.4, "Configuring Web Services in a Cluster"
- Section 9.5, "Monitoring Cluster Routing Performance"

> **Note:** For considerations specific to using Web service persistence in a cluster, see Section 7.4, "Using Web Service Persistence in a Cluster."

9.1 Overview of Web Services Cluster Routing

Clustering of *stateless* Web services—services that do not require knowledge of state information from prior invocations—is straightforward and works with existing WebLogic HTTP routing features on a third-party HTTP load balancer.

Clustering of Web services that require state information be maintained provides more challenges. Each instance of such a Web service is associated with state information that must be managed and persisted. The cluster routing decision is based on whether the message is bound to a specific server in the cluster. For example, if a particular server stores state information that is needed to process the message, and that state information is available only locally on that server.

> **Note:** Services that use session state replication to maintain their state are a separate class of problem from those that make use of advanced Web service features, such as Reliable Secure Profile. The latter require a more robust approach to persistence that may include storing state that may be available only from the local server. For more information, see Section , "A Note About Persistence."

In addition to ensuring that the Web service requests are routed to the appropriate server, the following general clustering requirements must be satisfied:

- The internal topology of a cluster must be transparent to clients. Clients interact with the cluster only through the front-end host, and do not need to be aware of

any particular server in the cluster. This enables the cluster to scale over time to meet the demands placed upon it.

- Cluster migration must be transparent to clients. Resources within the cluster (including persistent stores and other resources required by a Web service or Web service client) can be migrated from one server to another as the cluster evolves, responds to failures, and so on.

To meet the above requirements, the following methods are available for routing Web services in a cluster:

- **In-place SOAP router**—Assumes request messages arrive on the correct server and, if not, forwards the messages to the correct server ("at most one additional hop"). The routing decision is made by the Web service that receives the message. This routing strategy is the simplest to implement and requires no additional configuration. Though, it is not as robust as the next option.
- **Front-end SOAP router** (HTTP cluster servlet only)—Message routing is managed by the front-end host that accepts messages on behalf of the cluster and forwards them onto a selected member server of the cluster. For Web services, the front-end SOAP router inspects information in the SOAP message to determine the correct server to which it should route messages.

  This routing strategy is more complicated to configure, but is the most efficient since messages are routed directly to the appropriate server (avoiding any "additional hops").

> **Note:** When using Make Connection, as described in Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)", only front-end SOAP routing can guarantee proper routing of all messages related to a given Make Connection anonymous URI.

This chapter describes how to configure your environment to optimize the routing of Web services within a cluster. Use of the HTTP cluster servlet for the front-end SOAP router is described. The in-place SOAP router is also enabled and is used in the event the HTTP cluster servlet is not available or has not yet been initialized.

A Note About Persistence

While it is possible to maintain state for a Web service using the HttpSession as described in Section 21, "Programming Stateful JAX-WS Web Services Using HTTP Session," in some cases this simple persistence may not be robust enough. Advanced Web services features like reliable messaging, Make Connection, secure conversation, and so on, have robust persistence requirements that cannot be met by using the HttpSession alone. Advanced Web service features use a dedicated persistence implementation based on the concept of a *logical store*. For more information, see Section 7, "Managing Web Service Persistence."

At this time, these two approaches to persistence of Web service state are not compatible with each other. If you choose to write a clustered stateful Web service using HttpSession persistence and then use the advanced Web service features from that service (either as a client or service), Oracle cannot guarantee correct operation of your service in a cluster. This is because HttpSession replication may make the HttpSession available on a different set of servers than are hosting the persistence for advanced Web service features.

9.2 Cluster Routing Scenarios

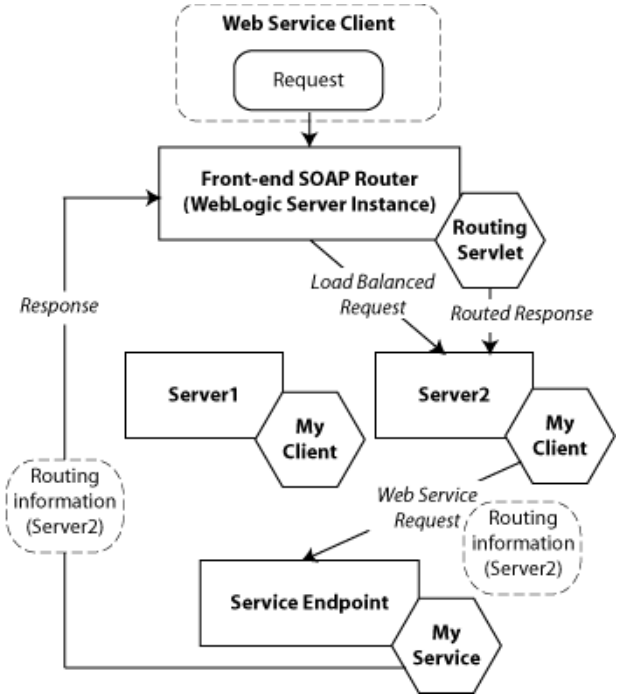
The following sections illustrate several scenarios for routing Web service request and response messages within a clustered environment:

- Section 9.2.1, "Scenario 1: Routing a Web Service Response to a Single Server"
- Section 9.2.2, "Scenario 2: Routing Web Service Requests to a Single Server Using Routing Information"
- Section 9.2.3, "Scenario 3: Routing Web Service Requests to a Single Server Using an ID"

9.2.1 Scenario 1: Routing a Web Service Response to a Single Server

In this scenario, an incoming request is load balanced to a server. Any responses to that request must be routed to that same server, which maintains state information on behalf of the original request.

Figure 9–1 Routing a Web Service Response to a Single Server



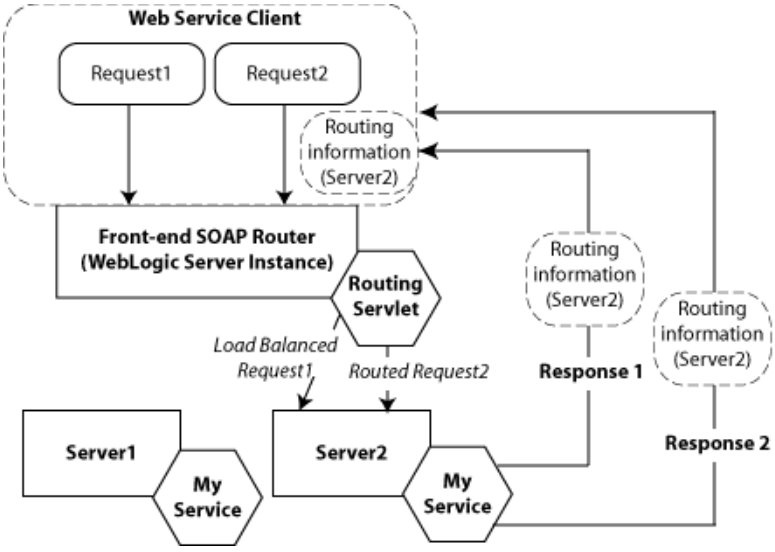
As shown in the previous figure:

1. The front-end SOAP router routes an incoming HTTP request and sends it to Server2 using standard load balancing techniques.
2. Server2 calls Myservice at the Web service endpoint address. The ReplyTo header in the SOAP message contains a pointer back to the front-end SOAP router.
3. MyService returns the response to the front-end SOAP router.
4. The front-end SOAP router must determine where to route the response. Because Server2 maintains state information that is relevant to the response, the front-end SOAP router routes the response to Server2.

9.2.2 Scenario 2: Routing Web Service Requests to a Single Server Using Routing Information

In this scenario, an incoming request is load balanced to a server. The response contains routing information that targets the original server for any subsequent requests

Figure 9–2 Routing Web Service Requests to a Single Server



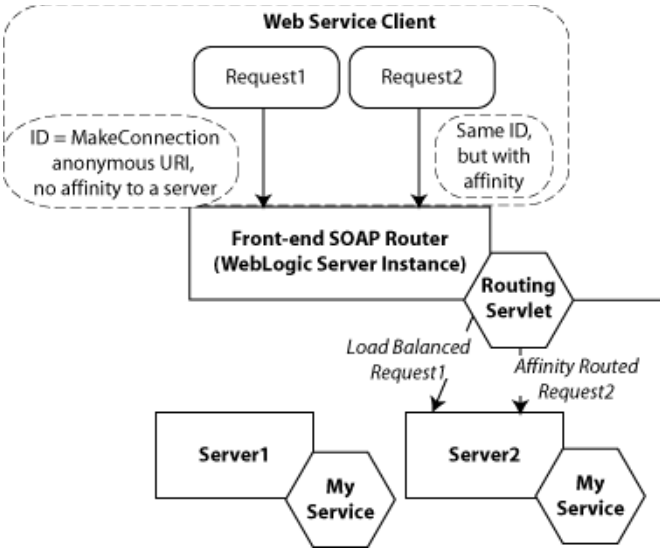
As shown in the previous figure:

1. The front-end SOAP router routes an incoming HTTP request (Request1) and sends it to Server 2 using standard load balancing techniques. The request has no routing information.
2. Server2 calls the Myservice at the Web service endpoint address. The ReplyTo header in the SOAP message contains a pointer back to the front-end SOAP router.
3. MyService returns the response to the caller. The response contains routing information that targets Server2 for any subsequent requests. The caller is responsible for passing the routing information contained in the response in any subsequent requests (for example, Request2).
4. The front-end SOAP router uses the routing information passed with Request2 to route the request to Server2.

9.2.3 Scenario 3: Routing Web Service Requests to a Single Server Using an ID

In this scenario, an incoming SOAP request contains an identifier, but no routing information. All subsequent requests with the same identifier must go to the same server.

Figure 9–3 Routing Web Service Requests to a Single Server Using an ID



As shown in the previous figure:

1. A request comes from a Web service client that includes an ID (Make Connection anonymous URI) that will be shared by future requests that are relevant to Request1. The form of this ID is protocol-specific.
2. The front-end SOAP router detects an ID in Request1 and checks the affinity store to determine if the ID is associated with a particular server in the cluster. In this case, there is no association defined.
3. The front-end SOAP router load balances the request and sends it to Server 2 for handling.
4. The MyService Web service instance on Server2 handles the request (generating a response, if required). Unlike in Section 9.2.2, "Scenario 2: Routing Web Service Requests to a Single Server Using Routing Information", routing information cannot be propagated in this case.
5. Request2 arrives at the front-end SOAP router using the same ID as that used in Request1.
6. The front-end SOAP router detects the ID and checks the affinity store to determine if the ID is associated with a particular server. This time, it determines that the ID is mapped to Server2.
7. Based on the affinity information, the front-end SOAP router routes Request2 to Server2.

9.3 How Web Service Cluster Routing Works

The following sections describe how Web service cluster routing works:

- Section 9.3.1, "Adding Routing Information to Outgoing Requests"
- Section 9.3.2, "Detecting Routing Information in Incoming Requests"
- Section 9.3.3, "Routing Requests Within the Cluster"
- Section 9.3.4, "Maintaining the Routing Map on the Front-end SOAP Router"

9.3.1 Adding Routing Information to Outgoing Requests

The Web services runtime adds routing information to the SOAP header of any outgoing message to ensure proper routing of messages in the following situations:

- The request is sent from a Web service client that uses a store that is not accessible from every member server in the cluster.
- The request requires in-memory state information used to process the response.

When processing an outgoing message, the Web services runtime:

- Creates a message ID for the outgoing request, if one has not already been assigned, and stores it in the `RelatesTo/MessageID` SOAP header using the following format:

  `uuid:WLS`*`format_version`*`:`*`store_name`*`:`*`uniqueID`*

  Where:

  - *`format_version`* specifies the WebLogic Server format version, for example `WLS1`.
  - *`store_name`* specifies the name of the persistent store, which specifies the store in use by the current Web service or Web service client sending the message. For example, `Server1Store`. This value may be a system-generated name, if the default persistent store is used, or an empty string if no persistent store is configured.
  - *`unique_ID`* specifies the unique message ID. For example: `68d6fc6f85a3c1cb:-2d3b89ab8:12068ad2e60:-7feb`

- Allows other Web service components to inject routing information into the message before it is sent.

9.3.2 Detecting Routing Information in Incoming Requests

The SOAP router (in-place or front-end) inspects incoming requests for routing information. In particular, the SOAP router looks for a `RelatesTo/MessageID` SOAP header to find the name of the persistent store and routes the message back to the server that hosts that persistent store.

In the event that there is an error in determining the correct server using front-end SOAP routing, then the message is sent to any server within the cluster and the in-place SOAP router is used. If in-place SOAP routing fails, then the sender of the message receives a fault on the protocol-specific back channel.

Note: SOAP message headers that contain routing information must be presented in clear text; they cannot be encrypted.

9.3.3 Routing Requests Within the Cluster

To assist in making a routing determination, the SOAP router (in-place or front-end) uses a dynamic map of store-to-server name associations. This dynamic map originates on the Managed Servers within a cluster and is accessed in memory by the in-place SOAP router and via HTTP response headers by the front-end SOAP router. The HTTP response headers are included automatically by WebLogic Server in every HTTP response sent by a Web service in the cluster.

Note: For more information about the HTTP response headers, see Section 9.3.4, "Maintaining the Routing Map on the Front-end SOAP Router".

Initially, the dynamic map is empty. It is only initialized after receiving its first response back from a Managed Server in the cluster. Until it receives back its first response with the HTTP response headers, the front-end SOAP router simply load balances the requests, and the in-place SOAP router routes the request to the appropriate server.

In the absence of SOAP-based routing information, it defers to the base routing that includes HTTP-session based routing backed by simple load balancing (for example, round-robin).

9.3.4 Maintaining the Routing Map on the Front-end SOAP Router

As noted in Section 9.3.3, "Routing Requests Within the Cluster", to assist in making a routing determination, the SOAP router (in-place or front-end) uses a dynamic map of store-to-server name associations.

To generate this dynamic map, two new HTTP response headers are provided, as described in the following sections. These headers are included automatically by WebLogic Server in every HTTP response sent by a Web service in the cluster.

Note: When implementing a third-party front-end to include the HTTP response headers described below, clients should send an HTTP request header with the following variable set to any value: `X-weblogic-wsee-request-storetoserver-list`

9.3.4.1 X-weblogic-wsee-storetoserver-list HTTP Response Header

A complete list of store-to-server mappings is maintained in the `X-weblogic-wsee-storetoserver-list` HTTP response header. The front-end SOAP router uses this header to populate a mapping that can be referenced at runtime to route messages.

The `X-weblogic-wsee-storetoserver-list` HTTP response header has the following format:

storename`1:`*host_server_spec* | *storename*`2:`*host_server_spec* | *storename*`3:`*host_server_spec*

In the above:

- *storename* specifies the name of the persistent store.
- *host_server_spec* is specifies using the following format: *servername:host:port:sslport*. If not known, the *sslport* is set to -1.

9.3.4.2 X-weblogic-wsee-storetoserver-hash HTTP Response Header

A hash mapping of the store-to-server list is provided in `X-weblogic-wsee-storetoserver-hash` HTTP response header. This header enables you to determine whether the new mapping list needs to be refreshed.

The `X-weblogic-wsee-storetoserver-hash` HTTP response header contains a String value representing the hash value of the list contained in the `X-weblogic-wsee-storetoserver-list` HTTP response header. By keeping

track of the last entry in the list, it can be determined whether the list needs to be refreshed.

9.4 Configuring Web Services in a Cluster

The following table summarizes the steps to configure Web services in a cluster.

Table 9–1 Steps to Manage Web Services in a Cluster

#	Step	Description
1	Set up the WebLogic cluster.	See Section 9.4.1, "Setting Up the WebLogic Cluster.".
2	Configure the clustered domain resources required for advanced Web service features.	You can configure automatically the clustered domain resources required using the cluster extension template script. Alternatively, you can configure the resources using the Oracle WebLogic Administration Console or WLST. See Section 9.4.2, "Configuring the Domain Resources Required for Web Service Advanced Features in a Clustered Environment.".
3	Extend the front-end SOAP router to support Web services.	**Note:** This step is required only if you are using the front-end SOAP router. The Web services routing servlet extends the functionality of the WebLogic HTTP cluster servlet to support routing of Web services in a cluster. See Section 9.4.3, "Extending the Front-end SOAP Router to Support Web Services.".
4	Enable routing of Web services atomic transaction messages.	See Section 9.4.4, "Enabling Routing of Web Services Atomic Transaction Messages."
5	Enable routing of Web services Make Connection messages.	See Section 9.4.5, "Enabling Routing of Web Services Make Connection Messages."
6	Configure the identity of the front-end SOAP router.	Each WebLogic Server instance in the cluster must be configured with the address and port of the front-end SOAP router. See Section 9.4.6, "Configuring the Identity of the Front-end SOAP Router.".

9.4.1 Setting Up the WebLogic Cluster

Set up the WebLogic cluster, as described in "Setting up WebLogic Clusters" in *Using Clusters for Oracle WebLogic Server*. Please note:

- To configure the clustered domain, see Section 9.4.2, "Configuring the Domain Resources Required for Web Service Advanced Features in a Clustered Environment."
- To enable SOAP-based front-end SOAP routing, configure an HTTP cluster servlet, as described in "Set Up the HttpClusterServlet" in *Using Clusters for Oracle WebLogic Server*.

9.4.2 Configuring the Domain Resources Required for Web Service Advanced Features in a Clustered Environment

When creating or extending a domain using Configuration Wizard, you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`) to configure automatically the resources required to support the advanced Web service features in a clustered environment. Although use of this extension template is not required, it makes the configuration of the required

resources much easier. Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Administration Console or WLST.

In addition, the template installs scripts into the domain directory that can be used to manage the resource required for advanced Web services in-sync as the domain evolves (for example, servers are added or removed, and so on).

For more information about how to configure the domain and run the scripts to manage resources, see "Configuring Your Domain for Advanced Web Service Features" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

9.4.3 Extending the Front-end SOAP Router to Support Web Services

Note: If you are not using the front-end SOAP router, then this step is not required.

You extend the front-end SOAP router to support Web services by specifying the `RoutingHandlerClassName` parameter shown in the following example (in **bold**), as part of the WebLogic HTTP cluster servlet definition.

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>HttpClusterServlet</servlet-name>
    <servlet-class>weblogic.servlet.proxy.HttpClusterServlet</servlet-class>
    <init-param>
       <param-name>WebLogicCluster</param-name>
       <param-value>Server1:7001|Server2:7001</param-value>
    </init-param>
   <init-param>
      <param-name>RoutingHandlerClassName</param-name>
      <param-value>
         weblogic.wsee.jaxws.cluster.proxy.SOAPRoutingHandler
      </param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>HttpClusterServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
. . .
</web-app>
```

9.4.4 Enabling Routing of Web Services Atomic Transaction Messages

High availability and routing of Web services atomic transaction messages is automatically enabled in Web service clustered environments. However, if the WebLogic HTTP cluster servlet is being used as the front-end server, you need to set the following system property to `false` on the server hosting the WebLogic HTTP cluster servlet:

```
weblogic.wsee.wstx.wsat.deployed=false
```

In addition, when using a WebLogic Server plugin, you should configure the `WLIOTimeoutSecs` parameter value appropriately. This parameter defines the

amount of time the plug-in waits for a response to a request from WebLogic Server. If the value is less than the time the servlets take to process, then you may see unexpected results. For more information about the `WLIOTimeoutSecs` parameter, see "General Parameters for Web Server Plug-ins" in *Using Web Server Plug-Ins with Oracle WebLogic Server*.

9.4.5 Enabling Routing of Web Services Make Connection Messages

To support Web Service Make Connection, as described in Section 4.6, "Using Asynchronous Web Service Clients From Behind a Firewall (Make Connection)," you must configure a default logical store on the WebLogic Server that is hosting the WebLogic HTTP cluster servlet. For information about configuring the default logical store, see Section 7.3.1, "Configuring the Logical Store."

9.4.6 Configuring the Identity of the Front-end SOAP Router

Each WebLogic Server instance in the cluster must be configured with the address and port of the front-end SOAP router. **Network channels** enable you to provide a consistent way to access the front-end address of a cluster. For more information about network channels, see "Understanding Network Channels" in *Configuring Server Environments for Oracle WebLogic Server*.

For each server instance:

1. Create a network channel for the protocol you use to invoke the Web service. You must name the network channel `weblogic-wsee-proxy-channel-`*`XXX`*, where *`XXX`* refers to the protocol. For example, to create a network channel for HTTPS, call it `weblogic-wsee-proxy-channel-https`.

   See "Configure custom network channels" in *Oracle WebLogic Server Administration Console Help* for general information about creating a network channel.

2. Configure the network channel, updating the **External Listen Address** and **External Listen Port** fields with the address and port of the proxy server, respectively.

9.5 Monitoring Cluster Routing Performance

You can monitor the following cluster routing statistics to evaluate the application performance:

- Total number of requests and responses.
- Total number of requests and responses that were routed specifically to the server.
- Routing failure information, including totals and last occurrence.

You can use the WebLogic Server Administration Console or WLST to monitor cluster routing performance. For information about using WebLogic Server Administration Console to monitor cluster routing performance, see "Monitor Web services" and "Monitor Web service clients,", in *Oracle WebLogic Server Administration Console Help*.

10 Using Web Services Atomic Transactions

This chapter describes how to use Web services atomic transactions for WebLogic Web services using Java API for XML Web Services (JAX-WS) to enable interoperability with other external transaction processing systems.

This chapter includes the following sections:

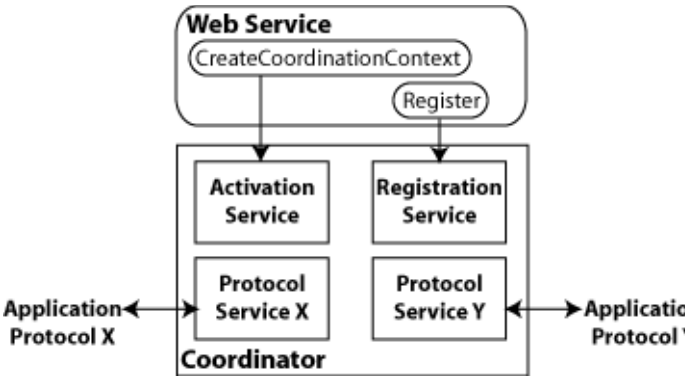
10.1 Overview of Web Services Atomic Transactions

WebLogic Web services enable interoperability with other external transaction processing systems, such as Websphere, JBoss, Microsoft .NET, and so on, through the support of the following specifications:

- Web Services Atomic Transaction (WS-AtomicTransaction) Versions 1.0, 1.1, and 1.2:
  `http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-cs-01/wstx-wsat-1.2-spec-cs-01.html`
- Web Services Coordination (WS-Coordination) Versions 1.0, 1.1, and 1.2:
  `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-cs-01/wstx-wscoor-1.2-spec-cs-01.html`

These specifications define an extensible framework for coordinating distributed activities among a set of participants. The **coordinator**, shown in the following figure, is the central component, managing the transactional state (coordination context) and enabling Web services and clients to register as participants.

Figure 10–1 Web Services Atomic Transactions Framework



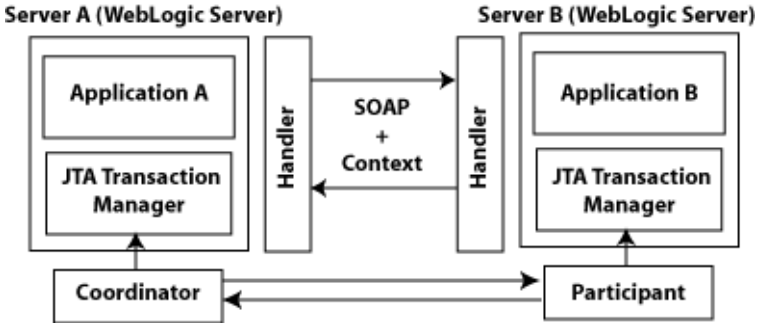
The following table describes the components of Web services atomic transactions, shown in the previous figure.

Table 10–1 Components of Web Services Atomic Transactions

Component	Description
Coordinator	Manages the transactional state (coordination context) and enables Web services and clients to register as participants.
Activation Service	Enables the application to activate a transaction and create a coordination context for an activity. Once created, the coordination context is passed with the transaction flow.
Registration Service	Enables an application to register as a participant.
Application Protocol X, Y	Supported coordination protocols, such as WS-AtomicTransaction.

The following figure shows two instances of WebLogic Server interacting within the context of a Web services atomic transaction. For simplicity, two WebLogic Web service applications are shown.

Figure 10–2 Web Services Atomic Transactions in WebLogic Server Environment



Please note the following:

- Using the local JTA transaction manager, a transaction can be imported to or exported from the local JTA environment as a *subordinate transaction*, all within the context of a Web service request.
- Creation and management of the coordination context is handled by the local JTA transaction manager.

- All transaction integrity management and recovery processing is done by the local JTA transaction manager.

For more information about JTA, see *Programming JTA for Oracle WebLogic Server*.

The following describes a sample end-to-end Web services atomic transaction interaction, illustrated in Figure 10–2:

1. Application A begins a transaction on the current thread of control using the JTA transaction manager on Server A.
2. Application A calls a Web service method in Application B on Server B.
3. Server A updates its transaction information and creates a SOAP header that contains the coordination context, and identifies the transaction and local coordinator.
4. Server B receives the request for Application B, detects that the header contains a transaction coordination context and determines whether it has already registered as a participant in this transaction. If it has, that transaction is resumed and if not, a new transaction is started.

   Application B executes within the context of the imported transaction. All transactional resources with which the application interacts are enlisted with this imported transaction.
5. Server B enlists itself as a participant in the WS-AtomicTransaction transaction by registering with the registration service indicated in the transaction coordination context.
6. Server A resumes the transaction.
7. Application A resumes processing and commits the transaction.

10.2 Configuring the Domain Resources Required for Web Service Advanced Features

When creating or extending a domain, if you expect that you will be using other Web service advanced features in addition to Web service atomic transactions (either now or in the future), you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`) to configure automatically the resources required to support the advanced Web service features. Although use of this extension template is not required, it makes the configuration of the required resources much easier. Alternatively, you can configure the resources required for these advanced features using the Oracle WebLogic Administration Console or WLST. For more information, see "Configuring Your Domain for Advanced Web Service Features" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Note: If you do not expect to use other Web service advanced features with Web service atomic transactions, application of this extension template is not required, minimizing start-up times and memory footprint.

10.3 Enabling Web Services Atomic Transactions on Web Services

To enable Web services atomic transactions on a Web service:

- When starting from Java (bottom-up), add the `@weblogic.wsee.wstx.wsat.Transactional` annotation to the Web service

endpoint implementation class or method. For more information, see Section 10.3.1, "Using the @Transactional Annotation in Your JWS File".

- When starting from WSDL (top-down), use `wsdlc` to generate a Web service from an existing WSDL file. In this case, The WS-AtomicTransaction policy assertions that are advertised in the WSDL are carried forward and are included in the WSDL file for the new Web service generated by `wsdlc`. See Section 10.3.2, "Enabling Web Services Atomic Transactions Starting From WSDL".
- At deployment time, enable and configure Web services atomic transactions at the Web service endpoint or method level using the WebLogic Server Administration Console. For more information, see Section 10.5, "Configuring Web Services Atomic Transactions Using the Administration Console".

The following tables summarizes the configuration options that you can set when enabling Web services atomic transactions.

Table 10–2 Web Services Atomic Transactions Configuration Options

Attribute	Description
`Version`	Version of the Web services atomic transaction coordination context that is used for Web services and clients. For clients, it specifies the version used for outbound messages only. The value specified must be consistent across the entire transaction. Valid values include `WSAT10`, `WSAT11`, `WSAT12`, and `DEFAULT`. The `DEFAULT` value for Web services is all three versions (driven by the inbound request); the `DEFAULT` value for Web service clients is `WSAT10`.
`Flow type`	Whether the Web services atomic transaction coordination context is passed with the transaction flow. For valid values, see Table 10–3.

The following table summarizes the valid values for flow type and their meaning on the Web service and client. The table also summarizes the valid value combinations when configuring web services atomic transactions for an EJB-style web service that uses the `@TransactionAttribute` annotation.

Table 10–3 Flow Types Values

Value	Web Service Client	Web Service	Valid EJB `@TransactionAttribute` Values
`NEVER`	**JTA transaction**: Do not export transaction coordination context. **No JTA transaction**: Do not export transaction coordination context.	**Transaction flow exists**: Do not import transaction coordination context. If the CoordinationContext header contains `mustunderstand="true"`, a SOAP fault is thrown. **No transaction flow**: Do not import transaction coordination context.	`NEVER`, `NOT_SUPPORTED`, `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`
`SUPPORTS` (Default)	**JTA transaction**: Export transaction coordination context. **No JTA transaction**: Do not export transaction coordination context.	**Transaction flow exists**: Import transaction context. **No transaction flow**: Do not import transaction coordination context.	`REQUIRED`, `SUPPORTS`
`MANDATORY`	**JTA transaction**: Export transaction coordination context. **No JTA transaction**: An exception is thrown.	**Transaction flow exists**: Import transaction context. **No transaction flow**: Service-side exception is thrown.	`MANDATORY`, `REQUIRED`, `SUPPORTS`

10.3.1 Using the @Transactional Annotation in Your JWS File

To enable Web services atomic transactions, specify the `@weblogic.wsee.wstx.wsat.Transactional` annotation on the Web service endpoint implementation class or method.

> **Note:** This annotation is not to be mistaken with `weblogic.jws.Transactional`, which ensures that the annotated class or operation runs inside of a transaction, but not an *atomic* transaction.

Please note the following:

- If you specify the `@Transactional` annotation at the Web service class level, the settings apply to all two-way methods defined by the service endpoint interface. You can override the flow type value at the method level; however, the version must be consistent across the entire transaction.
- You cannot explicitly specify the `@Transactional` annotation on a Web method that is also annotated with `@Oneway`.
- Web services atomic transactions cannot be used with the client-side asynchronous programming model.

The format for specifying the `@Transactional` annotation is as follows:

```
@Transactional(
     version=Transactional.Version.[WSAT10|WSAT11|WSAT12|DEFAULT],
     value=Transactional.TransactionFowType.[MANDATORY|SUPPORTS|NEVER]
)
```

For more information about the version and flow type configuration options, see Table 10–2.

The following sections provide examples of using the `@Transactional` annotation at the Web service implementation class and method levels, and with the EJB @TransactionAttribute annotation.

- Section 10.3.1.1, "Example: Using @Transactional Annotation on a Web Service Class"
- Section 10.3.1.2, "Example: Using @Transactional Annotation on a Web Service Method"
- Section 10.3.1.3, "Example: Using the @Transactional and the EJB @TransactionAttribute Annotations Together"

10.3.1.1 Example: Using @Transactional Annotation on a Web Service Class

The following example shows how to add `@Transactional` annotation on a Web service class. Relevant code is shown in **bold**. As shown in the example, there is an active JTA transaction.

> **Note:** The following excerpt is borrowed from the Web services atomic transaction example that is delivered with the WebLogic Server Samples Server. For more information, see Section 10.7, "More Examples of Using Web Services Atomic Transactions".

```
package examples.webservices.jaxws.wsat.simple.service;
. . .
import weblogic.jws.Policy;
import javax.transaction.UserTransaction;
. . .
import javax.jws.WebService;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;

/**
 * This JWS file forms the basis of a WebLogic WS-Atomic Transaction Web Service with the
 * operations: createAccount, deleteAccount, transferMonet, listAccount
 *
 */

@WebService(serviceName = "WsatBankTransferService", targetNamespace = "http://tempuri.org/",
            portName = "WSHttpBindingIService")
@Transactional(value=Transactional.TransactionFlowType.MANDATORY,
               version=weblogic.wsee.wstx.wsat.Transactional.Version.WSAT10)
public class WsatBankTransferService {

    public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;
        UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-demoXA-2");
                String sql = "insert into wsat_acct_remote (acctno, amount) values (" + acctNo +
                           ", " + amount + ")";
```

```
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
            throw new java.lang.Exception("insert fail at remote.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating. ";
            } catch (SQLException e) {
              System.out.println("**** Exception caught *****");
               e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at remote.");
        }
        } catch (java.lang.Exception e) {
          System.out.println("**** Exception caught *****");
          e.printStackTrace();
          throw new java.lang.Exception(e);
        }
    }
    public String deleteAccount(String acctNo) throws java.lang.Exception{
      ...
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
       java.lang.Exception{
      ...
    }
    public String listAccount() throws java.lang.Exception{
      ...
    }
}
```

10.3.1.2 Example: Using @Transactional Annotation on a Web Service Method

The following example shows how to add `@Transactional` annotation on a Web service implementation method. Relevant code is shown in **bold**.

```
package examples.webservices.jaxws.wsat.simple.service;
. . .
import weblogic.jws.Policy;
import javax.transaction.UserTransaction;
. . .
import javax.jws.WebService;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;

/**
 * This JWS file forms the basis of a WebLogic WS-Atomic Transaction Web Service with the
 * operations: createAccount, deleteAccount, transferMonet, listAccount
 *
 */

@WebService(serviceName = "WsatBankTransferService", targetNamespace = "http://tempuri.org/",
            portName = "WSHttpBindingIService")
public class WsatBankTransferService {

@Transactional(value=Transactional.TransactionFlowType.MANDATORY,
               version=weblogic.wsee.wstx.wsat.Transactional.Version.WSAT10)
    public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;
        UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
```

```
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-demoXA-2");
                String sql = "insert into wsat_acct_remote (acctno, amount) values (" + acctNo +
                           ", " + amount + ")";
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
            throw new java.lang.Exception("insert fail at remote.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating. ";
            } catch (SQLException e) {
              System.out.println("**** Exception caught *****");
               e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at remote.");
        }
        } catch (java.lang.Exception e) {
          System.out.println("**** Exception caught *****");
          e.printStackTrace();
          throw new java.lang.Exception(e);
        }
    }
    public String deleteAccount(String acctNo) throws java.lang.Exception{
      ...
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
       java.lang.Exception{
      ...
    }
    public String listAccount() throws java.lang.Exception{
      ...
    }
}
```

10.3.1.3 Example: Using the @Transactional and the EJB @TransactionAttribute Annotations Together

The following example illustrates how to use the `@Transactional` and EJB `@TransactionAttribute` annotations together. In this case, the flow type values must be compatible, as outlined in Table 10–3. Relevant code is shown in **bold**.

```
package examples.webservices.jaxws.wsat.simple.service;
. . .
import weblogic.jws.Policy;
import javax.transaction.UserTransaction;
. . .
import javax.jws.WebService;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import weblogic.wsee.wstx.wsat.Transactional;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;

/**
 * This JWS file forms the basis of a WebLogic WS-Atomic Transaction Web Service with the
 * operations: createAccount, deleteAccount, transferMonet, listAccount
 *
 */

@WebService(serviceName = "WsatBankTransferService", targetNamespace = "http://tempuri.org/",
            portName = "WSHttpBindingIService")
@Transactional(value=Transactional.TransactionFlowType.MANDATORY,
```

```
                version=weblogic.wsee.wstx.wsat.Transactional.Version.WSAT10)
@TransactionAttribute(TransactionAttributeType.REQUIRED
public class WsatBankTransferService {
. . .
}
```

10.3.2 Enabling Web Services Atomic Transactions Starting From WSDL

When enabled, Web services atomic transactions are advertised in the WSDL file using a policy assertion.

Table 10–4 summarizes the WS-AtomicTransaction 1.2 policy assertions that correspond to a set of common Web services atomic transaction flow type and EJB Transaction attribute combinations. All other combinations result in a build-time error.

Table 10–4 Web Services Atomic Transaction Policy Assertion Values (WS-AtomicTransaction 1.2)

Atomic Transaction Flow Type	EJB `@TransactionAttribute`	WS-AtomicTransaction 1.2 Policy Assertion
`MANDATORY`	`MANDATORY`, `REQUIRED`, `SUPPORTS`	`<wsat:ATAssertion/>`
`SUPPORTS`	`REQUIRED`, `SUPPORTS`	`<wsat:ATAssertion wsp:Optional="true"/>`
`NEVER`	`REQUIRED`, `REQUIRES_NEW`, `NEVER`, `SUPPORTS`, `NOT_SUPPORTED`	No policy advertisement

You can use `wsdlc` Ant task to generate, from an existing WSDL file, a set of artifacts that together provide a partial Java implementation of the Web service described by the WSDL file. The WS-AtomicTransaction policy assertions that are advertised in the WSDL are carried forward and are included in the WSDL file for the new Web service generated by `wsdlc`.

The `wsdlc` Ant tasks creates a JWS file that contains a partial (stubbed-out) implementation of the generated JWS interface. You need to modify this file to include your business code. After you have coded the JWS file with your business logic, run the `jwsc` Ant task to generate a complete Java implementation of the Web service. Use the `compiledWsdl` attribute of `jwsc` to specify the JAR file generated by the `wsdlc` Ant task which contains the JWS interface file and data binding artifacts. By specifying this attribute, the `jwsc` Ant task does not generate a new WSDL file but instead uses the one in the JAR file. Consequently, when you deploy the Web service and view its WSDL, the deployed WSDL will look just like the one from which you initially started (with the WS-AtomicTransaction policy assertions).

For complete details about using `wsdlc` to generate a Web service from a WSDL file, see "Developing WebLogic Web Services Starting From a WSDL File: Main Steps" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

10.4 Enabling Web Services Atomic Transactions on Web Service Clients

On a Web service client, enable Web services atomic transactions using one of the following methods:

- Add the `@weblogic.wsee.wstx.wsat.Transactional` annotation on the Web service reference injection point for a client. For more information, see

Section 10.4.1, "Using @Transactional Annotation with the @WebServiceRef Annotation".

- Pass an instance of the `weblogic.wsee.wstx.wsat.TransactionalFeature` as a parameter when creating the Web service proxy or dispatch. For more information, see Section 10.4.2, "Passing the TransactionalFeature to the Client".
- At deployment time, enable and configure Web services atomic transactions at the Web service client endpoint or method level using the WebLogic Server Administration Console. For more information, see Section 10.5, "Configuring Web Services Atomic Transactions Using the Administration Console".
- At run-time, if the non-atomic transactional Web service client calls an atomic transaction-enabled Web service, then based on the flow type advertised in the WSDL:
  - If the flow type is set to `SUPPORTS` or `NEVER` on the service-side, then the call is included as part of the transaction.
  - If the flow type is set to `MANDATORY`, then an exception is thrown.

For information about the configuration options that you can set when enabling Web services atomic transactions, see Table 10–2.

10.4.1 Using @Transactional Annotation with the @WebServiceRef Annotation

To enable Web services atomic transactions, specify the `@weblogic.wsee.wstx.wsat.Transactional` annotation on the Web service client at the Web service reference (`@WebServiceRef`) injection point.

The format for specifying the `@Transactional` annotation is as follows:

```
@Transactional(
     version=Transactional.Version.[WSAT10|WSAT11|WSAT12|DEFAULT],
     value=Transactional.TransactionFlowType.[MANDATORY|SUPPORTS|NEVER]
)
```

For more information about the version and flow type configuration options, see Table 10–2.

The following example illustrates how to annotate the Web service reference injection point. Relevant code is shown in **bold**. As shown in the example, the active JTA transaction becomes a part of the atomic transaction.

> **Note:** The following excerpt is borrowed from the Web services atomic transaction example that is delivered with the WebLogic Server Samples Server. For more information, see Section 10.7, "More Examples of Using Web Services Atomic Transactions".

```
package examples.webservices.jaxws.wsat.simple.client;
. . .
import javax.servlet.*;
import javax.servlet.http.*;
. . .
import java.net.URL;
import javax.xml.namespace.QName;

import javax.transaction.UserTransaction;
import javax.transaction.SystemException;
```

```
import javax.xml.ws.WebServiceRef;
import weblogic.wsee.wstx.wsat.Transactional;
*/

/**
 * This example demonstrates using a WS-Atomic Transaction to create or delete an account,
 * or transfer money via Web service as a single atomic transaction.
 */

public class WsatBankTransferServlet extends HttpServlet {
. . .
        String url = "http://localhost:7001";
        URL wsdlURL = new URL(url + "/WsatBankTransferService/WsatBankTransferService");
. . .
        DataSource ds = null;
        UserTransaction utx = null;

        try {
            ctx = new InitialContext();
            utx = (UserTransaction) ctx.lookup("javax.transaction.UserTransaction");
            utx.setTransactionTimeout(900);
        } catch (java.lang.Exception e) {
            e.printStackTrace();
        }

        WsatBankTransferService port = getWebService(wsdlURL);

        try {
            utx.begin();
            if (remoteAccountNo.length() > 0) {
                if (action.equals("create")) {
                    result = port.createAccount(remoteAccountNo, amount);
                } else if (action.equals("delete")) {
                    result = port.deleteAccount(remoteAccountNo);
                } else if (action.equals("transfer")) {
                    result = port.transferMoney(remoteAccountNo, amount, direction);
                }
            }
            utx.commit();
            result = "The transaction is committed " + result;
       } catch (java.lang.Exception e) {
           try {
                  e.printStackTrace();
                  utx.rollback();
                  result = "The transaction is rolled back.    " + e.getMessage();
           } catch(java.lang.Exception ex) {
             e.printStackTrace();
             result = "Exception is caught. Check stack trace.";
           }
      }
      request.setAttribute("result", result);
 . . .
    @Transactional(value = Transactional.TransactionFlowType.MANDATORY,
                   version = Transactional.Version.WSAT10)
     @WebServiceRef(wsdlLocation =
          "http://localhost:7001/WsatBankTransferService/WsatBankTransferService?WSDL", value =
          examples.webservices.jaxws.wsat.simple.service.WsatBankTransferService.class)
    WsatBankTransferService_Service service;
    private WsatBankTransferService getWebService() {
          return service.getWSHttpBindingIService();
```

```
    }

    public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;
         UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-dataSource-demoXAPool");
                String sql = "insert into wsat_acct_local (acctno, amount) values (
                    " + acctNo + ", " + amount + ")";
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
                    throw new java.lang.Exception("insert fail at local.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating.. ";
            } catch (SQLException e) {
                System.out.println("**** Exception caught *****");
                e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at local.");
            }
        } catch (java.lang.Exception e) {
        System.out.println("**** Exception caught *****");
            e.printStackTrace();
        throw new java.lang.Exception(e);
        }
    }

    public String deleteAccount(String acctNo) throws java.lang.Exception{
    . . .
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
         java.lang.Exception{
    . . .
    }
    public String listAccount() throws java.lang.Exception{
    . . .
    }
}
```

10.4.2 Passing the TransactionalFeature to the Client

To enable Web services atomic transactions on the client of the Web service, you can pass an instance of the `weblogic.wsee.wstx.wsat.TransactionalFeature` as a parameter when creating the Web service proxy or dispatch, as illustrated in the following example. Relevant code is shown in **bold**.

> **Note:** The following excerpt is borrowed from the Web services atomic transaction example that is delivered with the WebLogic Server Samples Server. For more information, see Section 10.7, "More Examples of Using Web Services Atomic Transactions".

```
package examples.webservices.jaxws.wsat.simple.client;
. . .
import javax.servlet.*;
import javax.servlet.http.*;
. . .
```

```
import java.net.URL;
import javax.xml.namespace.QName;

import javax.transaction.UserTransaction;
import javax.transaction.SystemException;

import weblogic.wsee.wstx.wsat.TransactionalFeature;
import weblogic.wsee.wstx.wsat.Transactional.Version;
import weblogic.wsee.wstx.wsat.Transactional.TransactionFlowType;
*/

/**
 * This example demonstrates using a WS-Atomic Transaction to create or delete an account,
 * or transfer money via Web service as a single atomic transaction.
 */

public class WsatBankTransferServlet extends HttpServlet {
. . .
        String url = "http://localhost:7001";
        URL wsdlURL = new URL(url + "/WsatBankTransferService/WsatBankTransferService");
. . .
        DataSource ds = null;
        UserTransaction utx = null;

        try {
            ctx = new InitialContext();
            utx = (UserTransaction) ctx.lookup("javax.transaction.UserTransaction");
            utx.setTransactionTimeout(900);
        } catch (java.lang.Exception e) {
            e.printStackTrace();
        }

        WsatBankTransferService port = getWebService(wsdlURL);

        try {
            utx.begin();
            if (remoteAccountNo.length() > 0) {
                if (action.equals("create")) {
                    result = port.createAccount(remoteAccountNo, amount);
                } else if (action.equals("delete")) {
                    result = port.deleteAccount(remoteAccountNo);
                } else if (action.equals("transfer")) {
                    result = port.transferMoney(remoteAccountNo, amount, direction);
                }
            }
            utx.commit();
            result = "The transaction is committed " + result;
       } catch (java.lang.Exception e) {
           try {
                  e.printStackTrace();
                  utx.rollback();
                  result = "The transaction is rolled back.    " + e.getMessage();
           } catch(java.lang.Exception ex) {
             e.printStackTrace();
             result = "Exception is caught. Check stack trace.";
           }
      }
      request.setAttribute("result", result);
 . . .
    // Passing the TransactionalFeature to the Client
```

```
    private WsatBankTransferService getWebService(URL wsdlURL) {
        TransactionalFeature feature = new TransactionalFeature();
        feature.setFlowType(TransactionFlowType.MANDATORY);
        feature.setVersion(Version.WSAT10);
        WsatBankTransferService_Service service = new WsatBankTransferService_Service(wsdlURL,
              new QName("http://tempuri.org/", "WsatBankTransferService"));
        return service.getWSHttpBindingIService(new javax.xml.ws.soap.AddressingFeature(),
              feature);
    }

    public String createAccount(String acctNo, String amount) throws java.lang.Exception{
        Context ctx = null;
         UserTransaction tx = null;
        try {
            ctx = new InitialContext();
            tx = (UserTransaction)ctx.lookup("javax.transaction.UserTransaction");
            try {
                DataSource dataSource = (DataSource)ctx.lookup("examples-dataSource-demoXAPool");
                String sql = "insert into wsat_acct_local (acctno, amount) values (
                    " + acctNo + ", " + amount + ")";
                int insCount = dataSource.getConnection().prepareStatement(sql).executeUpdate();
                if (insCount != 1)
                    throw new java.lang.Exception("insert fail at local.");
                return ":acctno=" + acctNo + " amount=" + amount + " creating.. ";
            } catch (SQLException e) {
                System.out.println("**** Exception caught *****");
                e.printStackTrace();
               throw new SQLException("SQL Exception during createAccount() at local.");
            }
        } catch (java.lang.Exception e) {
        System.out.println("**** Exception caught *****");
            e.printStackTrace();
        throw new java.lang.Exception(e);
        }
    }

    public String deleteAccount(String acctNo) throws java.lang.Exception{
    . . .
    }
    public String transferMoney(String acctNo, String amount, String direction) throws
         java.lang.Exception{
    . . .
    }
    public String listAccount() throws java.lang.Exception{
    . . .
    }
}
```

10.5 Configuring Web Services Atomic Transactions Using the Administration Console

The following sections describe how to configure Web services atomic transactions using the Administration Console.

- Section 10.5.1, "Securing Messages Exchanged Between the Coordinator and Participant"
- Section 10.5.2, "Enabling and Configuring Web Services Atomic Transactions"

10.5.1 Securing Messages Exchanged Between the Coordinator and Participant

Using transport-level security, you can secure messages exchanged between the Web services atomic transaction coordinator and participant by configuring the properties defined in the following table using the WebLogic Server Administration Console. These properties are configured at the domain level. For detailed steps, see "Configure Web services atomic transactions" in the *Oracle WebLogic Server Administration Console Help*.

Table 10–5 Securing Web Services Atomic Transactions

Property	Description
Web Services Transactions Transport Security Mode	Specifies whether two-way SSL is used for the message exchange between the coordinator and participant. This property can be set to one of the following values: ■ SSL Not Required—All Web service transaction protocol messages are exchanged over the HTTP channel. ■ SSL Required—All Web service transaction protocol messages are exchanged over the HTTPS channel. This flag must be enabled when invoking Microsoft .NET Web services that have atomic transactions enabled. ■ Client Certificate Required—All Web service transaction protocol messages are exchanged over HTTPS and a client certificate is required. For more information, see "Configure two-way SSL" in the *Oracle WebLogic Server Administration Console Help*.
Web Service Transactions Issued Token Enabled	Flag the specifies whether to use an issued token to enable authentication between the coordinator and participant. The `IssuedToken` is issued by the coordinator and consists of a security context token (SCT) and a session key used for signing. The participant sends the signature, signed using the shared session key, in its registration message. The coordinator authenticates the participant by verifying the signature using the session key.

10.5.2 Enabling and Configuring Web Services Atomic Transactions

To enable Web services atomic transactions and configure the version and flow type, you can customize the configuration at the endpoint or method level for the Web service or client. For detailed steps, see "Configure Web services atomic transactions" in the *Oracle WebLogic Server Administration Console Help*.

10.6 Using Web Services Atomic Transactions in a Clustered Environment

For considerations when using atomic transaction-enabled Web services in a clustered environment, see Chapter 9, "Managing Web Services in a Cluster".

10.7 More Examples of Using Web Services Atomic Transactions

Refer to the following sections for additional examples of using Web services atomic transactions:

- For an example of how to sign and encrypt message headers exchanged during the Web services atomic transaction, see "Securing Web Services Atomic Transactions" in *Securing WebLogic Web Services for Oracle WebLogic Server*.

  Note: You can secure applications that enable Web service atomic transactions using *only* WebLogic Web service security policies. You cannot secure them using Oracle Web Services Manager (WSM) policies.

- A detailed example of Web services atomic transactions is provided as part of the WebLogic Server sample application. For more information about running the sample application and accessing the example, see "Sample Application and Code Examples" in *Understanding Oracle WebLogic Server*.

11

Publishing a Web Service Endpoint

This chapter describes how to create a Web service endpoint at runtime *without deploying* the Web service to a WebLogic Server instance using the `javax.xml.ws.Endpoint` API.

For more information, see `http://download.oracle.com/javaee/5/api/javax/xml/ws/Endpoint.html`.

The following table summarizes the steps to publish a Web service endpoint.

Table 11–1 Steps to Publish a Web Service Endpoint

#	Step	Description
1	Create a Web service endpoint.	Use the `javax.xml.ws.Endpoint create()` method to create the endpoint, specify the *implementor* (that is, the Web service implementation) to which the endpoint is associated, and optionally specify the binding type. If not specified, the binding type defaults to `SOAP1.1/HTTP`. The endpoint is associated with only one implementation object and one `javax.xml.ws.Binding`, as defined at runtime; these values cannot be changed. For example, the following example creates a Web service endpoint for the `CallbackWS()` implementation. `Endpoint callbackImpl = Endpoint.create(new CallbackWS());`
2	Publish the Web service endpoint to accept incoming requests.	Use the `javax.xml.ws.Endpoint publish()` method to specify the server context, or the address and optionally the implementor of the Web service endpoint. **Note:** If you wish to update the metadata documents (WSDL or XML schema) associated with the endpoint, you must do so before publishing the endpoint. For example, the following example publishes the Web service endpoint created in Step 1 using the server context. `Object sc context.getMessageContext().get(MessageContext.SERVLET_CONTEXT); callbackImpl.publish(sc);`

Table 11–1 (Cont.) Steps to Publish a Web Service Endpoint

#	Step	Description
3	Stop the Web service endpoint to shut it down and prevent additional requests after processing is complete.	Use the `javax.xml.ws.Endpoint stop()` method to shut down the endpoint and stop accepting incoming requests. Once stopped, an endpoint cannot be republished. For example: `callbackImpl.stop()`

For an example of publishing a Web service endpoint within the context of a callback example, see Section 12.5, "Programming Guidelines for the Callback Client Web Service".

In addition to the steps described in the previous table, you can defined the following using the `javax.xml.ws.Endpoint` API methods:

- Endpoint metadata documents (WSDL or XML schema) associated with the endpoint. You must define metadata before publishing the Web service endpoint.
- Endpoint properties.
- `java.util.concurrent.Executor` that will be used to dispatch incoming requests to the application (see `http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Executor.html`).

For more information, see the `javax.xml.ws.Endpoint` Javadoc at `http://download.oracle.com/javaee/5/api/javax/xml/ws/Endpoint.html`.

12 Using Callbacks

This chapter describes how to use callbacks to notify clients of events for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

12.1 Overview of Callbacks

A callback is a contract between a client and service that allows the service to invoke operations on a client-provided endpoint during the invocation of a service method for the purpose of querying the client for additional data, allowing the client to inject behavior, or notifying the client of progress. The service advertises the requirements for the callback using a WSDL that defines the callback port type and the client informs the service of the callback endpoint address using WS-Addressing.

12.2 Example Callback Implementation

The example callback implementation described in this section consists of the following three Java files:

- **JWS file that implements the *callback Web service***: The callback Web service defines the callback methods. The implementation simply passes information back to the target Web service that, in turn, passes the information back to the client Web service.

  In the example in this section, the callback Web service is called `CallbackService`. The Web service defines a single callback method called `callback()`.

- **JWS file that implements the *target Web service***: The target Web service includes one or more standard operations that invoke a method defined in the callback Web service and sends the message back to the client Web service that originally invoked the operation of the target Web service.

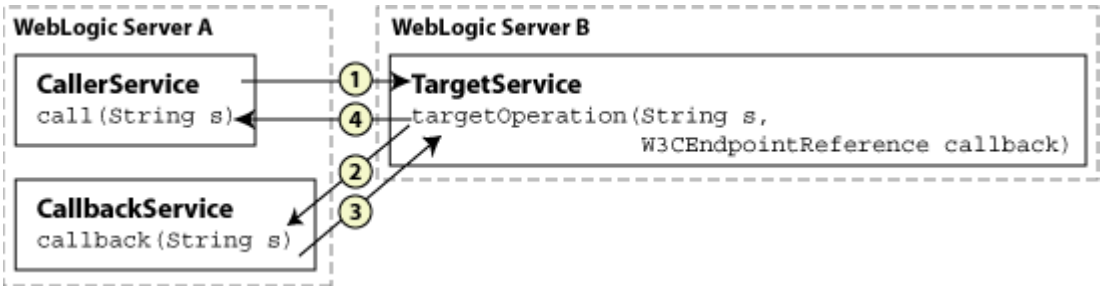
In the example, this Web service is called `TargetService` and it defines a single standard method called `targetOperation()`.

- **JWS file that implements the *client Web service***: The client Web service invokes an operation of the target Web service. Often, this Web service will include one or more methods that specify what the client should do when it receives a callback message back from the target Web service via a callback method.

  In the example, this Web service is called `CallerService`. The method that invokes `TargetService` in the standard way is called `call()`.

The following shows the flow of messages for the example callback implementation.

Figure 12–1 Example Callback Implementation



1. The `call()` method of the `CallerService` Web service, running in one WebLogic Server instance, explicitly invokes the `targetOperation()` method of the `TargetService` and passes a Web service endpoint to the `CallbackService`. Typically, the `TargetService` service is running in a separate WebLogic Server instance.
2. The implementation of the `TargetService.targetOperation()` method explicitly invokes the `callback()` method of the `CallbackService`, which implements the callback service, using the Web service endpoint that is passed in from `CallerService` when the method is called.
3. The `CallbackService.callback()` method sends information back to the `TargetService` Web service.
4. The `TargetService.targetOperation()` method, in turn, sends the information back to the `CallerService` service, completing the callback sequence.

12.3 Steps to Program Callbacks

The procedure in this section describes how to program and compile the three JWS files that are required to implement callbacks: the target Web service, the client Web service, and the callback Web service. The procedure shows how to create the JWS files from scratch; if you want to update existing JWS files, you can also use this procedure as a guide.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the Web services. For more information, see *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 12–1 Steps to Program Callbacks

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the target Web service.	Use your favorite IDE or text editor. See Section 12.4, "Programming Guidelines for Target Web Service". **Note:** The JWS file that implements the target Web service invokes one or more callback methods of the callback Web service. However, the step that describes how to program the callback Web service comes later in this procedure. For this reason, programmers typically program the three JWS files at the same time, rather than linearly as implied by this procedure. The steps are listed in this order for clarity only.
2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the target JWS file into a Web service.	See Section 12.7, "Updating the build.xml File for the Target Web Service".
3	Run the Ant target to build the target Web service.	For example: `prompt> ant build-target`
4	Deploy the target Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
5	Create a new JWS file, or update an existing one, that implements the client Web service.	It is assumed that the client Web service is deployed to a different WebLogic Server instance from the one that hosts the target Web service. See Section 12.5, "Programming Guidelines for the Callback Client Web Service".
6	Create the JWS file that implements the callback Web service.	See Section 12.6, "Programming Guidelines for the Callback Web Service".

7	Update the `build.xml` file that builds the client Web service.	The `jwsc` Ant task that builds the client Web service also compiles `CallbackWS.java` and includes the class file in the WAR file using the `Fileset` Ant task element. For example:
		`<clientgen`
		`type="JAXWS"`
		`wsdl="${awsdl}"`
		`packageName="jaxws.callback.client.add"/>`
		`<clientgen`
		`type="JAXWS"`
		`wsdl="${twsdl}"`
		`packageName="jaxws.callback.client.target"/>`
		`<FileSet dir="." >`
		`<include name="CallbackWS.java" />`
		`</FileSet>`

8	Run the Ant target to build the client and callback Web services.	For example: `prompt> ant build-caller`
9	Deploy the client Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

12.4 Programming Guidelines for Target Web Service

The following example shows a simple JWS file that implements the target Web service; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.callback;

import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.wsaddressing.W3CEndpointReference;
import examples.webservices.callback.callbackservice.*;

@WebService(
        portName="TargetPort",
        serviceName="TargetService",
        targetNamespace="http://example.oracle.com",
        endpointInterface=
                   "examples.webservices.callback.target.TargetPortType",
        wsdlLocation="/wsdls/Target.wsdl")
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http")

public class TargetImpl {
  public String targetOperation(String s, W3CEndpointReference callback)
  {
      CallbackService aservice = new CallbackService();
      CallbackPortType aport =
         aservice.getPort(callback, CallbackPortType.class);
      String result = aport.callback(s);
      return result + " processed by target";
  }
}
```

Follow these guidelines when programming the JWS file that implements the target Web service. Code snippets of the guidelines are shown in **bold** in the preceding example.

- Import the packages required to pass the callback service endpoint and access the `CallbackService` stub implementation.

  ```
  import javax.xml.ws.wsaddressing.W3CEndpointReference;
  import examples.webservices.callback.callbackservice.*;
  ```

- Create an instance of the `CallbackService` implementation using the stub implementation and get a port by passing the `CallbackService` service endpoint, which is passed by the calling application (`CallerService`).

  ```
  CallbackService aservice = new CallbackService();
  CallbackPortType aport =
       aservice.getPort(callback, CallbackPortType.class);
  ```

- Invoke the callback operation of `CallbackService` using the port you instantiated:

  ```
  String result = aport.callback(s);
  ```

- Return the result to the `CallerService` service.

  ```
  return result + " processed by target";
  ```

12.5 Programming Guidelines for the Callback Client Web Service

The following example shows a simple JWS file for a client Web service that invokes the target Web service described in Section 12.4, "Programming Guidelines for Target Web Service"; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.callback;

import javax.annotation.Resource;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.Endpoint;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.WebServiceRef;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.wsaddressing.W3CEndpointReference;

import examples.webservices.callback.target.*;

@WebService(
        portName="CallerPort",
        serviceName="CallerService",
        targetNamespace="http://example.oracle.com")
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http")

public class CallerImpl
{
   @Resource
   private WebServiceContext context;

   @WebServiceRef()
   private TargetService target;

   @WebMethod()
   public String call(String s) {
      Object sc =
          context.getMessageContext().get(MessageContext.SERVLET_CONTEXT);
      Endpoint callbackImpl = Endpoint.create(new CallbackWS());
      callbackImpl.publish(sc);
      TargetPortType tPort = target.getTargetPort();
      String result = tPort.targetOperation(s,
           callbackImpl.getEndpointReference(W3CEndpointReference.class));
       callbackImpl.stop();
      return result;
  }
}
```

Follow these guidelines when programming the JWS file that invokes the target Web service; code snippets of the guidelines are shown in **bold** in the preceding example:

- Import the packages required to access the servlet context, publish the Web service endpoint, and access the `TargetService` stub implementation.

  ```
  import javax.xml.ws.Endpoint;
  import javax.xml.ws.WebServiceContext;
  import javax.xml.ws.handler.MessageContext;
  import javax.xml.ws.wsaddressing.W3CEndpointReference;
  import examples.webservices.callback.target.*;
  ```

- Get the servlet context using the `WebServiceContext` and `MessageContext`. You will use the servlet context when publishing the Web service endpoint, later.

```
@Resource
private WebServiceContext context;
.
.
.
Object sc
        context.getMessageContext().get(MessageContext.SERVLET_CONTEXT);
```

  For more information about accessing runtime information using `WebServiceContext` and `MessageContext`, see "Accessing Runtime Information About a Web service" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

- Create a Web service endpoint to the `CallbackService` implementation and publish that endpoint to accept incoming requests.

```
Endpoint callbackImpl = Endpoint.create(new CallbackWS());
callbackImpl.publish(sc);
```

  For more information about Web service publishing, see Chapter 11, "Publishing a Web Service Endpoint."

- Access an instance of the `TargetService` stub implementation and invoke the `targetOperation` operation of `TargetService` using the port you instantiated. You pass the `CallbackService` service endpoint as a `javax.xml.ws.wsaddressing.W3CEndpointReference` data type:

```
@WebServiceRef()
private TargetService target;
.
.
.
TargetPortType tPort = target.getTargetPort();
String result = tPort.targetOperation(s,
    callbackImpl.getEndpointReference(W3CEndpointReference.class));
```

- Stop publishing the endpoint:

```
callbackImpl.stop();
```

12.6 Programming Guidelines for the Callback Web Service

The following example shows a simple JWS file for a callback Web service. The `callback` operation is shown in **bold**.

```
package examples.webservices.callback;

import javax.jws.WebService;
import javax.xml.ws.BindingType;

@WebService(
     portName="CallbackPort",
     serviceName="CallbackService",
     targetNamespace="http://example.oracle.com",
     endpointInterface=
         "examples.webservices.callback.callbackservice.CallbackPortType",
     wsdlLocation="/wsdls/Callback.wsdl")
```

```
@BindingType(value="http://schemas.xmlsoap.org/wsdl/soap/http")

public class CallbackWS implements
     examples.webservices.callback.callbackservice.CallbackPortType {

     public CallbackWS() {
     }

     public java.lang.String callback(java.lang.String arg0) {
          return arg0.toUpperCase();
     }
}
```

12.7 Updating the build.xml File for the Target Web Service

You update a `build.xml` file to generate a target Web service that invokes the callback Web service by adding `taskdefs` and a `build-target` target that looks something like the following example. See the description after the example for details.

```
<taskdef name="jwsc"
  classname="weblogic.wsee.tools.anttasks.JwscTask" />

<target name="build-target">
  <jwsc srcdir="src" destdir="${ear-dir}" listfiles="true">
       <jws file="TargetImpl.java"
         compiledWsdl="${cowDir}/target/Target_wsdl.jar" type="JAXWS">
         <WLHttpTransport contextPath="target" serviceUri="TargetService"/>
       </jws>
       <clientgen
         type="JAXWS"
         wsdl="Callback.wsdl"
         packageName="examples.webservices.callback.callbackservice"/>
  </jwsc>
   <zip destfile="${ear-dir}/jws.war" update="true">
      <zipfileset dir="src/examples/webservices/callback" prefix="wsdls">
         <include name="Callback*.wsdl"/>
      </zipfileset>
   </zip>
 </target>
```

Use the `taskdef` Ant task to define the full classname of the `jwsc` Ant tasks. Update the `jwsc` Ant task that compiles the client Web service to include:

- `<clientgen>` child element of the `<jws>` element to generate and compile the `Service` interface stubs for the deployed `CallbackService` Web service. The `jwsc` Ant task automatically packages them in the generated WAR file so that the client Web service can immediately access the stubs. You do this because the `TartgetImpl` JWS file imports and uses one of the generated classes.
- `<zip>` element to include the WSDL for the `CallbackService` service in the WAR file so that other Web services can access the WSDL from the following URL: `http://${wls.hostname}:${wls.port}/callback/wsdls/Callback.wsdl`.

For more information about `jwsc`, see "Running the jwsc WebLogic Web Services Ant Task" in *Getting Started With JAX-RPC Web Services for Oracle WebLogic Server*.

13

Optimizing Binary Data Transmission

This chapter describes how to send SOAP messages as attachments to optimize transmission for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

13.1 Optimizing Binary Data Transmission Using MTOM/XOP

SOAP Message Transmission Optimization Mechanism/XML-binary Optimized Packaging (MTOM/XOP) defines a method for optimizing the transmission of XML data of type `xs:base64Binary` or `xs:hexBinary` in SOAP messages. When the transport protocol is HTTP, Multipurpose Internet Mail Extension (MIME) attachments are used to carry that data while at the same time allowing both the sender and the receiver direct access to the XML data in the SOAP message without having to be aware that any MIME artifacts were used to marshal the `base64Binary` or `hexBinary` data.

The binary data optimization process involves the following steps:

1. Encode the binary data.
2. Remove the binary data from the SOAP envelope.
3. Compress the binary data.
4. Attach the binary data to the MIME package.
5. Add references to the MIME package in the SOAP envelope.

MTOM/XOP support is standard in JAX-WS via the use of JWS annotations. The MTOM specification does not require that, when MTOM is enabled, the Web service runtime use XOP binary optimization when transmitting `base64binary` or `hexBinary` data. Rather, the specification allows the runtime to choose to do so. This is because in certain cases the runtime may decide that it is more efficient to send the binary data directly in the SOAP Message; an example of such a case is when transporting small amounts of data in which the overhead of conversion and transport consumes more resources than just inlining the data as is.

The following Java types are mapped to the `base64Binary` XML data type, by default: `javax.activation.DataHandler`, `java.awt.Image`, and

`javax.xml.transform.Source`. The elements of type `base64Binary` or `hexBinary` are mapped to `byte[]`, by default.

The following table summarizes the steps required to use MTOM/XOP to send `base64Binary` or `hexBinary` attachments.

Table 13–1 Steps to Use MTOM/XOP to Send Binary Data

#	Step	Description
1	Annotate the data types that you are going to use as an MTOM attachment. (Optional)	Depending on your programming model, you can annotate your Java class or WSDL to define the content types that are used for sending binary data. This step is optional. By default, XML binary types are mapped to Java `byte[]`. For more information, see Section 13.1.1, "Annotating the Data Types".
2	Enable MTOM on the Web service.	See Section 13.1.2, "Enabling MTOM on the Web Service".
3	Enable MTOM on the client of the Web service.	See Section 13.1.3, "Enabling MTOM on the Client".
4	Set the attachment threshold.	Set the attachment threshold to specify when the `xs:binary64` data is sent inline or as an attachment. See Section 13.1.4, "Setting the Attachment Threshold".

For more information see:

- MTOM specification at `http://www.w3.org/TR/soap12-mtom`
- XOP specification at `http://www.w3.org/TR/xop10`

13.1.1 Annotating the Data Types

Depending on your programming model, you can annotate your Java class or WSDL to define the MIME content types that are used for sending binary data. This step is optional.

The following table defines the mapping of MIME content types to Java types. In some cases, a default MIME type-to-Java type mapping exists. If no default exists, the MIME content types are mapped to `DataHandler`.

Table 13–2 Mapping of MIME Content Types to Java Types

MIME Content Type	Java Type
`image/gif`	`java.awt.Image`
`image/jpeg`	`java.awt.Image`
`text/plain`	`java.lang.String`
`text/xml or application/xml`	`javax.xml.transform.Source`
`*/*`	`javax.activation.DataHandler`

The following sections describe how to annotate the data types based on whether you are starting from Java or WSDL.

- Section 13.1.1.1, "Annotating the Data Types: Start From Java"
- Section 13.1.1.2, "Annotating the Data Types: Start From WSDL"

13.1.1.1 Annotating the Data Types: Start From Java

When starting from Java, to define the content types that are used for sending binary data, annotate the field that holds the binary data using the `@XmlMimeType` annotation.

The field that contains the binary data must be of type `DataHandler`.

The following example shows how to annotate a field in the Java class that holds the binary data.

```
@WebMethod
@Oneway
public void dataUpload(
    @XmlMimeType("application/octet-stream") DataHandler data)
{
}
```

13.1.1.2 Annotating the Data Types: Start From WSDL

When starting from WSDL, to define the content types that are used for sending binary data, annotate the WSDL element of type `xs:base64Binary` or `xs:hexBinary` using one of the following attributes:

- `xmime:contentType` - Defines the content type of the element.
- `xmime:expectedContentType` - Defines the range of media types that are acceptable for the binary data.

The following example maps the `image` element of type `base64binary` to `image/gif` MIME type (which maps to the `java.awt.Image` Java type).

```
<element name="image" type="base64Binary"
xmime:expectedContentTypes="image/gif"
xmlns:xmime="http://www.w3.org/2005/05/xmlmime"/>
```

13.1.2 Enabling MTOM on the Web Service

You can enable MTOM on the Web service using an annotation or WS-Policy file, as described in the following sections:

- Enabling MTOM on the Web Service Using Annotation
- Enabling MTOM on the Web Services Using WS-Policy File

13.1.2.1 Enabling MTOM on the Web Service Using Annotation

To enable MTOM in the Web service, specify the `@java.xml.ws.soap.MTOM` annotation on the service endpoint implementation class, as illustrated in the following example. Relevant code is shown in **bold**.

```
package examples.webservices.mtom;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.soap.MTOM;

@MTOM
@WebService(name="MtomPortType",
            serviceName="MtomService",
            targetNamespace="http://example.org")
public class MTOMImpl {
  @WebMethod
  public String echoBinaryAsString(byte[] bytes) {
```

```
    return new String(bytes);

  }
}
```

13.1.2.2 Enabling MTOM on the Web Services Using WS-Policy File

In addition to the @MTOM annotation, described in the previous section, support for MTOM/XOP in WebLogic JAX-WS Web services is implemented using the pre-packaged WS-Policy file `Mtom.xml`. WS-Policy files follow the *WS-Policy* specification, described at http://www.w3.org/TR/ws-policy; this specification provides a general purpose model and XML syntax to describe and communicate the policies of a Web service, in this case the use of MTOM/XOP to send binary data. The installation of the pre-packaged `Mtom.xml` WS-Policy file in the `types` section of the Web service WSDL is as follows (provided for your information only; you cannot change this file):

```
<wsp:Policy wsu:Id="myService_policy">
     <wsp:ExactlyOne>
         <wsp:All>
             <wsoma:OptimizedMimeSerialization

xmlns:wsoma="http://schemas.xmlsoap.org/ws/2004/09/policy/optimizedmimeserializati
on" />
         </wsp:All>
     </wsp:ExactlyOne>
 </wsp:Policy>
```

When you deploy the compiled JWS file to WebLogic Server, the dynamic WSDL will automatically contain the following snippet that references the MTOM WS-Policy file; the snippet indicates that the Web service uses MTOM/XOP:

```
<wsdl:binding name="BasicHttpBinding_IMtomTest"
                       type="i0:IMtomTest">
     <wsp:PolicyReference URI="#myService_policy" />
     <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
```

You can associate the `Mtom.xml` WS-Policy file with a Web service at development-time by specifying the `@Policy` metadata annotation in your JWS file. Be sure you also specify the `attachToWsdl=true` attribute to ensure that the dynamic WSDL includes the required reference to the `Mtom.xml` file; see the example below.

You can associate the `Mtom.xml` WS-Policy file with a Web service at deployment time by modifying the WSDL to add the Policy to the types section just before deployment.

In addition, you can attach the file at runtime using by the Administration Console; for details, see "Associate a WS-Policy file with a Web service" in the *Oracle WebLogic Server Administration Console Help*. This section describes how to use the JWS annotation.

The following simple JWS file example shows how to use the `@weblogic.jws.Policy` annotation in your JWS file to specify that the pre-packaged `Mtom.xml` file should be applied to your Web service (relevant code shown in **bold**):

```
package examples.webservices.mtom;
import javax.jws.WebMethod;
import javax.jws.WebService;
 import weblogic.jws.Policy;
@WebService(name="MtomPortType",
```

```
            serviceName="MtomService",
            targetNamespace="http://example.org")
@Policy(uri="policy:Mtom.xml", attachToWsdl=true)
public class MtomImpl {
  @WebMethod
   public String echoBinaryAsString(byte[] bytes) {
     return new String(bytes);
   }
```

13.1.3 Enabling MTOM on the Client

To enable MTOM on the client of the Web service, pass an instance of the `javax.xml.ws.soap.MTOMFeature` as a parameter when creating the Web service proxy or dispatch, as illustrated in the following example. Relevant code is shown in **bold**.

```
package examples.webservices.mtom.client;

import javax.xml.ws.soap.MTOMFeature;

public class Main {
  public static void main(String[] args) {
    String FOO = "FOO";
    MtomService service = new MtomService()
    MtomPortType port = service.getMtomPortTypePort(new MTOMFeature());
    String result = null;
    result = port.echoBinaryAsString(FOO.getBytes());
    System.out.println( "Got result: " + result );
  }
}
```

13.1.4 Setting the Attachment Threshold

You can set the attachment threshold to specify when the `xs:binary64` data is sent inline or as an attachment. By default, the attachment threshold is 0 bytes. All `xs:binary64` data is sent as an attachment.

To set the attachment threshold:

- On the Web service, pass the `threshold` attribute to the `@java.xml.ws.soap.MTOM` annotation. For example:

  ```
  @MTOM(threshold=3072)
  ```

- On the client of the Web service, pass the threshold value to `javax.xml.ws.soap.MTOMFeature`. For example:

  ```
  MtomPortType port = service.getMtomPortTypePort(new MTOMFeature(3072));
  ```

In each of the examples above, if a message is greater than or equal to 3 KB, it will be sent as an attachment. Otherwise, the content will be sent inline, as part of the SOAP message body.

13.2 Streaming SOAP Attachments

> **Note:** The `com.sun.xml.ws.developer.StreamingDataHandler` API is supported as an extension to the JAX-WS RI. Because this API is not provided as part of the WebLogic software, it is subject to change.

Using MTOM and the `javax.activation.DataHandler` and `com.sun.xml.ws.developer.StreamingDataHandler` APIs you can specify that a Web service use a streaming API when reading inbound SOAP messages that include attachments, rather than the default behavior in which the service reads the entire message into memory. This feature increases the performance of Web services whose SOAP messages are particularly large.

> **Note:** Streaming MTOM cannot be used in conjunction with message encryption.

The following sections describe how to employ streaming SOAP attachments on the client and server sides.

13.2.1 Client Side Example

The following provides an example that employs streaming SOAP attachments on the client side.

```
package examples.webservices.mtomstreaming.client;

import java.util.Map;
import java.io.InputStream;
import javax.xml.ws.soap.MTOMFeature;
import javax.activation.DataHandler;
import javax.xml.ws.BindingProvider;
import com.sun.xml.ws.developer.JAXWSProperties;
import com.sun.xml.ws.developer.StreamingDataHandler;

public class Main {
   public static void main(String[] args) {
      MtomStreamingService service = new MtomStreamingService();
      MTOMFeature feature = new MTOMFeature();
      MtomStreamingPortType port = service.getMtomStreamingPortTypePort(
         feature);
      Map<String, Object> ctxt=((BindingProvider)port).getRequestContext();
      ctxt.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8192);
      DataHandler dh = new DataHandler(new
                  FileDataSource("/tmp/example.jar"));
      port.fileUpload("/tmp/tmp.jar",dh);

      DataHandler dhn = port.fileDownload("/tmp/tmp.jar");
      StreamingDataHandler sdh = {StreamingDataHandler)dh;
      try{
         File file = new File("/tmp/tmp.jar");
         sdh.moveTo(file);
         sdh.close();
       }
       catch(Exception e){
```

```
            e.printStackTrace();
         }
      }
   }
```

The preceding example demonstrates the following:

- To enable MTOM on the client of the Web service, pass an instance of the `javax.xml.ws.soap.MTOMFeature` as a parameter when creating the Web service proxy or dispatch.
- Configure HTTP streaming support by enabling HTTP chunking on the MTOM streaming client.

  ```
  Map<String, Object> ctxt = ((BindingProvider)port).getRequestContext();
     ctxt.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8192);
  ```

- Call the `port.fileUpload` method.
- Cast the `DataHandler` to `StreamingDataHandler` and use the `StreamingDataHandler.readOnce()` method to read the attachment.

13.2.2 Server Side Example

The following provides an example that employs streaming SOAP attachments on the server side.

```
package examples.webservices.mtomstreaming;

import java.io.File;
import java.jws.Oneway;
import javax.jws.WebMethod;
import java.io.InputStream;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlMimeType;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.soap.MTOM;
import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import com.sun.xml.ws.developer.StreamingAttachment;
import com.sun.xml.ws.developer.StreamingDataHandler;

@StreamingAttachment(parseEagerly=true, memoryThreshold=40000L)
@MTOM
@WebService(name="MtomStreaming",
            serviceName="MtomStreamingService",
            targetNamespace="http://example.org",
            wsdlLocation="StreamingImplService.wsdl")
@Oneway
@WebMethod
public class StreamingImpl {

   // Use @XmlMimeType to map to DataHandler on the client side
   public void fileUpload(String fileName,
                          @XmlMimeType("application/octet-stream")
                          DataHandler data) {
      try {
            StreamingDataHandler dh = (StreamingDataHandler) data;
            File file = new File(fileName);
            dh.moveTo(file);
            dh.close();
```

```
      } catch (Exception e) {
            throw new WebServiceException(e);
    }

  @XmlMimeType("application/octet-stream")
  @WebMethod
  public DataHandler fileDownload(String filename)
  {
    return  new DataHandler(new FileDataSource(filename));
  }
}
```

The preceding example demonstrates the following:

- The `@StreamingAttachement` annotation is used to configure the streaming SOAP attachment. For more information, see "Configuring Streaming SOAP Attachments" on page 13-8.
- The `@XmlMimeType` annotation is used to map the `DataHandler`, as follows:
  - If starting from WSDL, it is used to map the `xmime:expectedContentTypes="application/octet-stream"` to `DataHandler` in the generated SEI.
  - If starting from Java, it is used to generate an appropriate schema type in the generated WSDL.
- Cast the `DataHandler` to `StreamingDataHandler` and use the `StreamingDataHandler.moveTo(File)` method to store the contents of the attachment to a file.

13.2.3 Configuring Streaming SOAP Attachments

You can configure streaming SOAP attachments on the client and server sides to specify the following:

- Directory in which large attachments are stored.
- Whether to parse eagerly the streaming attachments.
- Maximum attachment size (bytes) that can be stored in memory. Attachments that exceed the specified number of bytes are written to a file.

13.2.3.1 Configuring Streaming SOAP Attachments on the Server

> **Note:** The `com.sun.xml.ws.developer.StreamingAttachment` API is supported as an extension to the JDK 6.0. Because this API is not provided as part of the JDK 6.0 kit, it is subject to change.

To configure streaming SOAP attachments on the server, add the `@StreamingAttachment` annotation on the endpoint implementation. The following example specifies that streaming attachments are to be parsed eagerly (read or write the complete attachment) and sets the memory threshold to 4MB. Attachments under 4MB are stored in memory.

```
...
import com.sun.xml.ws.developer.StreamingAttachment;
import javax.jws.WebService;
```

```
@StreamingAttachment(parseEagerly=true, memoryThreshold=4000000L)
@WebService(name="HelloWorldPortType", serviceName="HelloWorldService")
public class StreamingImpl {
}
```

13.2.3.2 Configuring Streaming SOAP Attachments on the Client

> **Note:** The `com.sun.xml.ws.developer.StreamingAttachmentFeature` API is supported as an extension to the JDK 6.0. Because this API is not provided as part of the JDK 6.0 kit, it is subject to change.

To configure streaming SOAP attachments on the client, create a `StreamingAttachmentFeature` object and pass this as an argument when creating the `PortType` stub implementation. The following example sets the directory in which large attachments are stored to `/tmp`, specifies that streaming attachments are to be parsed eagerly and sets the memory threshold to 4MB. Attachments under 4MB are stored in memory.

```
...
import com.sun.xml.ws.developer.StreamingAttachmentFeature;
...
MTOMFeature mtom = new MTOMFeature();
StreamingAttachmentFeature stf = new StreamingAttachmentFeature("/tmp", true,
4000000L);
MtomStreamingService service = new MtomStreamingService();
MtomStreamingPortType port = service.getMtomStreamingPortTypePort(
        mtom, stf);
...
```

13.3 Sending SOAP Messages With Attachments Using swaRef

Together, the specifications defined in Table 13–3 define a mechanism for sending SOAP messages with attachments using the `swaRef` XML attachment type.

Table 13–3 Specifications Supported for Sending SOAP Messages With Attachments

Specification	Description
SOAP With Attachments (SwA)	Defines a MIME `multipart/related` structure for packaging attachments with SOAP messages. For more information, see `http://www.w3.org/TR/SOAP-attachments`
WS-I Attachments Profile	Defines the `swaRef` schema type that can be used in the WSDL description to represent a reference to an attachment as a content-ID (CID) URL. WS-I publishes a public schema which defines the `swaRef` type, as defined by the following XSD: `http://ws-i.org/profiles/basic/1.1/xsd/swaref.xsd` JAXB maps the `swaRef` schema type to `javax.activation.DataHandler`. For more information, see: `http://www.ws-i.org/Profiles/AttachmentsProfile-1.0-2004-08-24.html`

The following shows an example of how to use `swaRef` in a WSDL file to specify that the `claimForm` request and response messages be passed as an attachment.

Example 13–1 Example of WSDL File Using swaRef Data Type

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions name="SOAPBuilders-mime-cr-test"
    xmlns:types="http://example.org/mime/data"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://example.org/mime"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    targetNamespace="http://example.org/mime">

    <wsdl:types>
        <schema
            xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://example.org/mime/data"
            xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
            elementFormDefault="qualified"
            xmlns:ref="http://ws-i.org/profiles/basic/1.1/xsd">
        <import namespace="http://ws-i.org/profiles/basic/1.1/xsd"
            schemaLocation="WS-ISwA.xsd"/>
. . .
        <complexType name="claimFormTypeRequest">
            <sequence>
                <element name="request" type="ref:swaRef"/>
            </sequence>
        </complexType>
        <complexType name="claimFormTypeResponse">
            <sequence>
                <element name="response" type="ref:swaRef"/>
            </sequence>
        </complexType>

        <element name="claimFormRequest"  type="types:claimFormTypeRequest"/>
        <element name="claimFormResponse" type="types:claimFormTypeResponse"/>
        </schema>

    </wsdl:types>
. . .
     <wsdl:message name="claimFormIn">
        <wsdl:part name="data" element="types:claimFormRequest"/>
    </wsdl:message>

    <wsdl:message name="claimFormOut">
        <wsdl:part name="data" element="types:claimFormResponse"/>
    </wsdl:message>
. . .
    <wsdl:portType name="Hello">
. . .
        <wsdl:operation name="claimForm">
            <wsdl:input message="tns:claimFormIn"/>
            <wsdl:output message="tns:claimFormOut"/>
        </wsdl:operation>
    </wsdl:portType>
. . .
</wsdl:definitions>
```

As specified in the WSDL example in Example 13–1, the XML content that is tagged as type `swaRef` is sent as a MIME attachment and the element inside the SOAP body holds the reference to this attachment, as shown in Example 13–2.

Example 13–2 Example of SOAP Message with MIME Attachment

```
Content-Type: Multipart/Related; start-info="text/xml"; type="application/xop+xml";
    boundary="----=_Part_4_32542424.1118953563492"Content-Length: 1193SOAPAction: ""
    ------=_Part_5_32550604.1118953563502Content-Type: application/xop+xml; type="text/xml";
charset=utf-8
    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
       <soapenv:Body>
       <request xmlns="http://example.org/mtom/data">
          cid:b0a597fd-5ef7-4f0c-9d85-6666239f1d25@example.jaxws.sun.com
       </request>
       </soapenv:Body>
    </soapenv:Envelope>
------=_Part_5_32550604.1118953563502
    Content-Type: application/xmlContent-ID:
   <b0a597fd-5ef7-4f0c-9d85-6666239f1d25@example.jaxws.sun.com>
<?xml
    version="1.0" encoding="UTF-8"?><application xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocaption="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd" version="1.4">
    <display-name>Simple example of application</display-name>
    <description>Simple example</description>
    <module>
        <ejb>ejb1.jar</ejb>
    </module>
    <module>
        <ejb>ejb2.jar</ejb>
    </module>
    <module>
         <web> <web-uri>web.war</web-uri> <context-root>web</context-root></web>
     </module></application>
```

Example 13–3 shows a sample Web service that defines the `claimForm` operation. As defined in the WSDL, the request and response messages are sent as MIME attachments.

Example 13–3 Example Web Service

```
package mime.server;

import javax.jws.WebService;
import javax.xml.ws.Holder;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.activation.DataHandler;
import java.awt.*;
import java.io.ByteArrayInputStream;

@WebService (endpointInterface = "mime.server.Hello")
public class HelloImpl {
    ...
    public ClaimFormTypeResponse claimForm(ClaimFormTypeRequest data){
        ClaimFormTypeResponse resp = new ClaimFormTypeResponse();
        resp.setResponse(data.getRequest());
        return resp;
    }
    ...
}
```

Example 13–4 shows a sample Web service client that calls the `claimForm` operation. Note that the client request data that will be transmitted as an attachment is mapped to the `DataHandler` data type.

Example 13–4 Example Web Service Client With MIME Attachments

```
package mime.client;

import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;
import javax.activation.DataHandler;
import java.io.ByteArrayInputStream;
import java.awt.*;

public class MimeApp {
    public static void main (String[] args){
        try {
            Object port = new HelloService().getHelloPort ();
            testSwaref ((Hello)port);
        } catch (Exception ex) {
            ex.printStackTrace ();
        }
    }

    private static void testSwaref (Hello port) throws Exception{
        DataHandler claimForm = new DataHandler (new StreamSource(
                    new ByteArrayInputStream(sampleXML.getBytes())), "text/xml");
        ClaimFormTypeRequest req = new ClaimFormTypeRequest();
        req.setRequest(claimForm);
        ClaimFormTypeResponse resp = port.claimForm (req);
        DataHandler out = resp.getResponse();
        ...
    }

    private static final String sampleXML = "<?xml version=\"1.0\" encoding=\"UTF-8\" ?> \n" +
            "<NMEAstd>\n" +
            "<DevIdSentenceId>$GPRMC</DevIdSentenceId>\n" +
            "<Time>212949</Time>\n" +
            "<Navigation>A</Navigation>\n" +
            "<NorthOrSouth>4915.61N</NorthOrSouth>\n" +
            "<WestOrEast>12310.55W</WestOrEast>\n" +
            "<SpeedOnGround>000.0</SpeedOnGround>\n" +
            "<Course>360.0</Course>\n" +
            "<Date>030904</Date>\n" +
            "<MagneticVariation>020.3</MagneticVariation>\n" +
            "<MagneticPoleEastOrWest>E</MagneticPoleEastOrWest>\n" +
            "<ChecksumInHex>*6B</ChecksumInHex>\n" +
            "</NMEAstd>";
}
```

14

Developing Dynamic Proxy Clients

This chapter highlights the differences between static and dynamic proxy clients, and describes the steps to develop a dynamic proxy client for WebLogic Web services using Java API for XML Web Services (JAX-WS)

This chapter includes the following sections:

- Section 14.1, "Overview of Static Versus Dynamic Proxy Clients"
- Section 14.2, "Steps to Develop a Dynamic Proxy Client"
- Section 14.3, "Additional Considerations When Specifying WSDL Location"

14.1 Overview of Static Versus Dynamic Proxy Clients

Table 14–1 highlights the differences between static and dynamic proxy clients.

Table 14–1 Static Versus Dynamic Proxy Clients

Proxy Client Type	Description
Static proxy client	Compile and bind the Web service client at development time. This generates a *static stub* for the Web service client proxy. The source code for the generated static stub client relies on a specific service implementation. As a result, this option offers the least flexibility. For examples of static proxy clients, see: ■ "Invoking Web Service Clients" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* ■ Chapter 3, "Roadmaps for Developing Web Service Clients"
Dynamic proxy client	Compile nothing at development time. At runtime, the application retrieves and interprets the WSDL and dynamically constructs calls. A *dynamic proxy client* enables a Web service client to invoke a Web service based on a service endpoint interface (SEI) dynamically at run-time (without using `clientgen`). This option does not rely upon a specific service implementation, providing greater flexibility, but also a greater performance hit. The steps to develop a dynamic proxy client are described in Section 14.2, "Steps to Develop a Dynamic Proxy Client."

14.2 Steps to Develop a Dynamic Proxy Client

The steps to create a dynamic proxy client are outlined in the following table. For more information, see the `javax.xml.ws.Service` Javadoc at `http://download.oracle.com/javaee/5/api/javax/xml/ws/Service.html`.

Table 14–2 Steps to Create a Dynamic Proxy Client

#	Step	Description

1	Create the `javax.xml.ws.Service` instance.	Create the `Service` instance using the `Service.create` method.
		You must pass the service name and optionally the location of the WSDL document. The method details are as follows:
		`public static Service create (QName serviceName) throws javax.xml.ws.WebServiceException {}`
		`public static Service create (URL wsdlDocumentLocation, QName serviceName) throws javax.xml.ws.WebServiceException {}`
		For example:
		`URL wsdlLocation = new URL("http://example.org/my.wsdl");`
		`QName serviceName = new QName("http://example.org/sample", "MyService");`
		`Service s = Service.create(wsdlLocation, serviceName);`
		See Section 14.3, "Additional Considerations When Specifying WSDL Location" for additional usage information.

2	Create the proxy stub.	Use the `Service.getPort` method to create the proxy stub. You can use this stub to invoke operations on the target service endpoint.
		You must pass the service endpoint interface (SEI) and optionally the name of the port in the WSDL service description. The method details are as follows:
		`public <T> T getPort(QName portName, Class<T> serviceEndpointInterface) throws javax.xml.ws.WebServiceException {}`
		`public <T> T getPort(Class<T> serviceEndpointInterface) throws javax.xml.ws.WebServiceException {}`
		For example:
		`MyPort port = s.getPort(MyPort.class);`

14.3 Additional Considerations When Specifying WSDL Location

If you use HTTPS to get the Web service from the WSDL, and the hostname definition in the WebLogic Server SSL certificate does not equal the hostname of the peer HTTPS server or is not one of the following, the action fails with a hostname verification error:

- localhost
- 127.0.0.1
- hostname of localhost
- IP address of localhost

The hostname verification error is as follows:

```
EchoService service = new EchoService(https-wsdl, webservice-qName);
:
:
javax.xml.ws.WebServiceException: javax.net.ssl.SSLKeyException:
Security:090504 Certificate chain received from host.company.com - 10.167.194.63
failed hostname verification check. Certificate contained {....} but
check expected host.company.com
```

The recommended workaround is to use HTTP instead of HTTPS to get the Web service from a WSDL when creating the service, and your own hostname verifier code to verify the hostname after the service is created:

```
EchoService service = Service.create(http_wsdl, qname);
//get Port
```

```
EchoPort port = service.getPort(...);
//set self-defined hostname verifier
((BindingProvider) port).getRequestContext().put(
      com.sun.xml.ws.developer.JAXWSProperties.HOSTNAME_VERIFIER,
      new  MyHostNameVerifier());
/*
*/
```

Optionally, you can ignore hostname verification by setting the binding provider property:

```
((BindingProvider) port).getRequestContext().put(
      BindingProviderProperties.HOSTNAME_VERIFICATION_PROPERTY,
      "true");
```

However, if you must use HTTPS to get the Web service from the WSDL, there are several possible workarounds:

- Turn off hostname verification if you are using the WebLogic Server HTTPS connection. To do this, set the global system property to ignore hostname verification:

  ```
  weblogic.security.SSL.ignoreHostnameVerification=true
  ```

  The system property does not work for service creation if the connection is a JDK connection or other non-WebLogic Server connection.

- Set your own hostname verifier for the connection before you get the Web service from the WSDL, then use HTTPS to get the Web service from the WSDL:

  ```
  //set self-defined hostname verifier
  URL url = new URL(https_wsdl);
  HttpsURLConnection connection = (HttpsURLConnection)url.openConnection();
  connection.setHostnameVerifier(new MyHostNameVerifier());

  //then initiate the service
  EchoService service = Service.create(https_wsdl, qname);

  //get port and set self-defined hostname verifier to binding provider
  ...
  ```

For the workarounds in which you set your own hostname verifier, an example hostname verifier might be as follows:

```
public class MyHostnameVerifier implements HostnameVerifier {
    public boolean verify(String hostname, SSLSession session) {
            if (hostname.equals("the host you want"))
          return true;
           else
               return false;
    }
}
```

15 Using XML Catalogs

This chapter describes how to use XML catalogs with WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

15.1 Overview of XML Catalogs

An XML catalog enables your application to reference imported XML resources, such as WSDLs and XSDs, from a source that is different from that which is part of the description of the Web service. Redirecting the XML resources in this way may be required to improve performance or to ensure your application runs properly in your local environment.

For example, a WSDL may be accessible during client generation, but may no longer be accessible when the client is run. You may need to reference a resource that is local to or bundled with your application rather than a resource that is available over the network. Using an XML catalog file, you can specify the location of the WSDL that will be used by the Web service at runtime.

The following table summarizes how XML catalogs are supported in the WebLogic Server Ant tasks.

Table 15–1 Support for XML Catalogs in WebLogic Server Ant Tasks

Ant Task	Description
`clientgen`	Define and reference XML catalogs in one of the following ways: ▪ Use the `catalog` attribute to specify the name of the external XML catalog file. For more information, see Section 15.2.1, "Defining an External XML Catalog". ▪ Use the `<xmlcatalog>` child element to reference an embedded XML catalog file. For more information, see Section 15.2.2, "Embedding an XML Catalog". When you execute the `clientgen` Ant task to build the client (or the `jwsc` Ant task if the `clientgen` task is embedded), the `jax-ws-catalog.xml` file is generated and copied to the client runtime environment. The `jax-ws-catalog.xml` file contains the XML catalog(s) that are defined in the external XML catalog file(s) and/or embedded in the `build.xml` file. This file is copied, along with the referenced XML targets, to the `META-INF` or `WEB-INF` folder for Enterprise or Web applications, respectively. **Note:** The contents of the XML resources are not impacted during this process. You can disable the `jax-ws-catalog.xml` file from being copied to the client runtime environment, as described in Section 15.3, "Disabling XML Catalogs in the Client Runtime".
`wsdlc`	Define and reference XML catalogs in one of the following ways: ▪ Use the `catalog` attribute to specify the name of the external XML catalog file. For more information, see Section 15.2.1, "Defining an External XML Catalog". ▪ Use the `<xmlcatalog>` child element to reference an embedded XML catalog file. For more information, see Section 15.2.2, "Embedding an XML Catalog". When you execute the `wsdlc` Ant task, the XML resources are copied to the compiled WSDL JAR file or exploded directory.
`wsdlget`	Define and reference XML catalogs in one of the following ways: ▪ Use the `catalog` attribute to specify the name of the external XML catalog file. For more information, see Section 15.2.1, "Defining an External XML Catalog". ▪ Use the `<xmlcatalog>` child element to reference an embedded XML catalog file. For more information, see Section 15.2.2, "Embedding an XML Catalog". When you execute the `wsdlget` Ant task, the WSDL and imported resources are downloaded to the specified directory. **Note:** The contents of the XML resources are updated to reference the resources defined in the XML catalog(s).

The following sections describe how to:

- Define and reference an XML catalog to specify the XML resources that you want to redirect. See Section 15.2, "Defining and Referencing XML Catalogs",
- Disable XML catalogs in the client runtime. See Section 15.3, "Disabling XML Catalogs in the Client Runtime".
- Get a local copy of the WSDL and its imported XML resources using `wsdlget`. These files can be packaged with your application and referenced from within an XML catalog. See Section 15.4, "Getting a Local Copy of XML Resources".

For more information about XML catalogs, see the *Oasis XML Catalogs* specification at http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html.

15.2 Defining and Referencing XML Catalogs

You define an XML catalog and then reference it from the `clientgen` or `wsdlc` Ant task in your `build.xml` file in one of the following ways:

- **Define an external XML catalog** - Define an external XML catalog file and reference that file from the `clientgen` or `wsdlc` Ant tasks in your `build.xml` file using the `catalogs` attribute. For more information, see Section 15.2.1, "Defining an External XML Catalog".
- **Embed an XML catalog** - Embed the XML catalog directly in the `build.xml` file using the `<xmlcatalog>` element and reference it from the `clientgen` or `wsdlc` Ant tasks in your `build.xml` file using the `<xmlcatalog>` child element. For more information, see Section 15.2.2, "Embedding an XML Catalog".

In the event of a conflict, entries defined in an embedded XML catalog take precedence over those defined in an external XML catalog.

Note: You can use the `wsdlget` Ant task to get a local copy of the XML resources, as described in Section 15.3, "Disabling XML Catalogs in the Client Runtime".

15.2.1 Defining an External XML Catalog

To define an external XML catalog:

1. Create an external XML catalog file that defines the XML resources that you want to be redirected. See Section 15.2.1.1, "Creating an External XML Catalog File".
2. Reference the XML catalog file from the `clientgen` or `wsdlc` Ant task in your `build.xml` file using the `catalogs` attribute. See Section 15.2.1.2, "Referencing the External XML Catalog File".

Each step is described in more detail in the following sections.

15.2.1.1 Creating an External XML Catalog File

The `<catalog>` element is the root element of the XML catalog file and serves as the container for the XML catalog entities. To specify XML catalog entities, you can use the `system` or `public` elements, for example.

The following provides a sample XML catalog file:

```
<catalog xmln="urn:oasis:names:tc:entity:xmlns:xml:catalog"
    prefer="system">
    <system systemId="http://foo.org/hello?wsdl"
            uri="HelloService.wsdl" />
    <public publicId="ISO 8879:1986//ENTITIES Added Latin 1//EN"
            uri="wsdl/myApp/myApp.wsdl"/>
</catalog>
```

In the above example:

- The `<catalog>` root element defines the XML catalog namespace and sets the `prefer` attribute to `system` to specify that system matches are preferred.

- The <system> element associates a URI reference with a system identifier.
- The <public> element associates a URI reference with a public identifier.

For a complete description of the XML catalog file syntax, see the *Oasis XML Catalogs* specification at http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html.

15.2.1.2 Referencing the External XML Catalog File

To reference the XML catalog file from the clientgen or wsdlc Ant task in your build.xml file, use the catalogs attribute.

The following example shows how to reference an XML catalog file using clientgen. Relevant code lines are shown in **bold**.

```
<target name="clientgen">
<clientgen
     type="JAXWS"
     wsdl="${wsdl}"
     destDir="${clientclasses.dir}"
     packageName="xmlcatalog.jaxws.clientgen.client"
     catalog="wsdlcatalog.xml"/>
</clientgen>
</target>
```

15.2.2 Embedding an XML Catalog

To embed an XML catalog:

1. Create an embedded XML catalog in the build.xml file. See Section 15.2.2.1, "Creating an Embedded XML Catalog".
2. Reference the embedded XML catalog from the clientgen or wsdlc Ant task using the xmlcatalog child element. See Section 15.2.2.2, "Referencing an Embedded XML Catalog".

Each step is described in more detail in the following sections.

Note: In the event of a conflict, entries defined in an embedded XML catalog take precedence over those defined in an external XML catalog.

15.2.2.1 Creating an Embedded XML Catalog

The <xmlcatalog> element enables you to embed an XML catalog directly in the build.xml file. The following shows a sample of an embedded XML catalog in the build.xml file.

```
<xmlcatalog id="wsimportcatalog">
     <entity publicid="http://helloservice.org/types/HelloTypes.xsd"
             location="${basedir}/HelloTypes.xsd"/>
</xmlcatalog>
```

For a complete description of the embedded XML catalog syntax, see the *Oasis XML Catalogs* specification at http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html.

15.2.2.2 Referencing an Embedded XML Catalog

The <xmlcatalog> child element of the clientgen or wsdlc Ant tasks enables you to reference an embedded XML catalog. To specify the <xmlcatalog> element, use the following syntax:

```
<xmlcatalog refid="id"/>
```

The *id* referenced by the <xmlcatalog> child element must match the ID of the embedded XML catalog.

The following example shows how to reference an embedded XML catalog using clientgen. Relevant code lines are shown in **bold**.

```
<target name="clientgen">
<clientgen
     type="JAXWS"
     wsdl="${wsdl}"
     destDir="${clientclasses.dir}"
     packageName="xmlcatalog.jaxws.clientgen.client"
     catalog="wsdlcatalog.xml"/>
     <xmlcatalog refid="wsimportcatalog"/>
</clientgen>
</target>
<xmlcatalog id="wsimportcatalog">
     <entity publicid="http://helloservice.org/types/HelloTypes.xsd"
             location="${basedir}/HelloTypes.xsd"/>
</xmlcatalog>
```

15.3 Disabling XML Catalogs in the Client Runtime

By default, when you define and reference XML catalogs in your build.xml file, as described in Section 15.2, "Defining and Referencing XML Catalogs", when you execute the clientgen Ant task to build the client, the jax-ws-catalog.xml file is generated and copied to the client runtime environment. The jax-ws-catalog.xml file contains the XML catalog(s) that are defined in the external XML catalog file(s) and/or embedded in the build.xml file. This file is copied, along with the referenced XML targets, to the META-INF or WEB-INF folder for Enterprise or Web applications, respectively.

You can disable the generation of the XML catalog artifacts in the client runtime environment by setting the genRuntimeCatalog attribute of the clientgen to false. For example:

```
<clientgen
     type="JAXWS"
     wsdl="${wsdl}"
     destDir="${clientclasses.dir}"
     packageName="xmlcatalog.jaxws.clientgen.client"
     catalog="wsdlcatalog.xml"
     genRuntimeCatalog="false"/>
```

In this case, the jax-ws-catalog.xml file will not be copied to the runtime environment.

If you generated your client with the genRuntimeCatalog attribute set to false, to subsequently enable the XML catalogs in the client runtime, you will need to create the jax-ws-catalog.xml file manually and copy it to the META-INF or WEB-INF folder for Enterprise or Web applications, respectively. Ensure that the jax-ws-catalog.xml file contains all of the entries defined in the external XML catalog file(s) and/or embedded in the build.xml file.

15.4 Getting a Local Copy of XML Resources

The `wsdlget` Ant task enables you to get a local copy of XML resources, such as WSDL and XSD files. Then, you can refer to the local version of the XML resources using an XML catalog, as described in Section 15.2, "Defining and Referencing XML Catalogs".

The following excerpt from an Ant `build.xml` file shows how to use the `wsdlget` Ant task to download a WSDL and its XML resources. The XML resources will be saved to the `wsdl` folder in the directory from which the Ant task is run.

```
<target name="wsdlget"
     <wsdlget
          wsdl="http://host/service?wsdl"
          destDir="./wsdl/"
     />
</target>
```

16

Handling Exceptions Using SOAP Faults

This chapter describes how to handle exceptions that occur when a message is being processed using Simple Object Access Protocol (SOAP) faults for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

16.1 Overview of Exception Handling Using SOAP Faults

When a Web service request is being processed, if an error is encountered, the nature of the error needs to be communicated to the client, or sender of the request. Because clients can be written on a variety of platforms using different languages, there must exist a standard, platform-independent mechanism for communicating the error.

The SOAP specification (available at `http://www.w3.org/TR/soap/`) defines a standard, platform-independent way of describing the error within the SOAP message using a *SOAP fault*. In general, a SOAP fault is analogous to an application exception. SOAP faults are generated by receivers to report business logic errors or unexpected conditions.

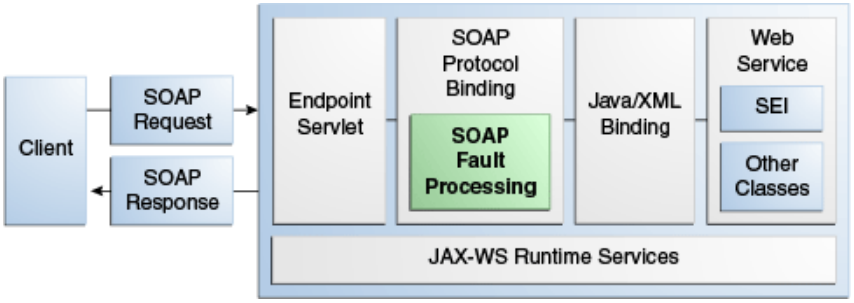
In JAX-WS, Java exceptions (`java.lang.Exception`) that are thrown by your Java Web service are mapped to a SOAP fault and returned to the client to communicate the reason for failure. SOAP faults can be one of the following types:

- Modeled—Maps to an exception that is thrown explicitly from the business logic of the Java code and mapped to `wsdl:fault` definitions in the WSDL file, when the Web service is deployed. In this case, the SOAP faults are predefined.
- Unmodeled—Maps to an exception (for example, `java.lang.RuntimeException`) that is generated at run-time when no business logic fault is defined in the WSDL. In this case, Java exceptions are represented as generic SOAP fault exceptions, `javax.xml.ws.soap.SOAPFaultException`.

The faults are returned to the sender only if request/response messaging is in use. If a Web service operation is configured as one-way, the SOAP fault is not returned to the sender, but stored for further processing.

As illustrated in Figure 16–1, JAX-WS handles SOAP fault processing during SOAP protocol binding. The SOAP binding maps exceptions to SOAP fault messages.

Figure 16–1 How SOAP Faults Are Processed



16.2 Contents of the SOAP Fault Element

The SOAP `<Fault>` element is used to transmit error and status information within a SOAP message. The `<Fault>` element is a child of the body element. There can be only one `<Fault>` element in the body of a SOAP message.

The SOAP `<Fault>` element contents for SOAP 1.2 and 1.1 are defined in the following sections:

- Section 16.2.1, "SOAP 1.2 <Fault> Element Contents"
- Section 16.2.2, "SOAP 1.1 <Fault> Element Contents"

16.2.1 SOAP 1.2 <Fault> Element Contents

The `<Fault>` element for SOAP 1.2 contains the subelements defined in Table 16–1.

Table 16–1 Subelements of the SOAP 1.2 <Fault> Element

Subelement	Description	Required?
`env:Code`	Information pertaining to the fault error code. The `env:Code` element consists of the following two subelements: ■ `env:Value` ■ `env: Subcode` The subelements are defined below.	Required
`env:Value`	Code value that provides more information about the fault. A set of code values is predefined by the SOAP specification, including: ■ `VersionMismatch`—Invalid namespace defined in SOAP envelope element. The SOAP envelope must conform to the `http://schemas.xmlsoap.org/soap/envelope` namespace. ■ `MustUnderstand`—SOAP header entry not understood by processing party. ■ `Sender`—Message was incorrectly formatted or is missing information. ■ `Receiver`—Problem with the server that prevented the message from being processed. ■ `DataEncodingUnknown`—Received message has an unrecognized encoding style value. You can define encoding styles for SOAP headerblocks and child elements of the SOAP body, and this encoding style must be recognized by the Web services server.	Required
`env:Subcode`	Subcode value that provides more information about the fault. This subelement can have a recursive structure.	Optional
`env:Reason`	Human-readable description of fault. The `<env:Reason>` element contains one or more `<Text>` elements, each of which contains information about the fault in a different language.	Required
`env:Node`	Information regarding the actor (SOAP node) that caused the fault.	Optional
`env:Role`	Role being performed by actor at the time of the fault.	Optional
`env:Detail`	Application-specific information, such as the exception that was thrown.	Optional

The following provides an example of a SOAP 1.2 fault message.

Example 16–1 Example of SOAP 1.2 Fault Message

```
<?xml version="1.0"?>
<env:Envelope xmlns:env=http://www.w3.org/2003/05/soap-envelope>
   <env:Body>
      <env:Fault>
         <env:Code>
            <env:Value>env:Sender</env:Value>
            <env:Subcode>
               <env:Value>rpc:BadArguments</env:Value>
            </env:Subcode>
         </env:Code>
         <env:Reason>
            <env:Text xml:lang=en-US>Processing error<env:Text>
         </env:Reason>
         <env:Detail>
            <e:myFaultDetails
               xmlns:e=http://travelcompany.example.org/faults>
               <e:message>Name does not match card number</e:message>
               <e:errorcode>999</e:errorcode>
            </e:myFaultDetails>
         </env:Detail>
```

```
      </env:Fault>
   </env:Body>
</env:Envelope>
```

16.2.2 SOAP 1.1 <Fault> Element Contents

The `<Fault>` element for SOAP 1.1 contains the subelements defined in Table 16–2.

Table 16–2 Subelements of the SOAP 1.1 <Fault> Element

Subelement	Description
`faultcode`	Standard code that provides more information about the fault. A set of code values is predefined by the SOAP specification, as defined below. This set of fault code values can be extended by the application. Predefined fault code values include: ■ `VersionMismatch`—Invalid namespace defined in SOAP envelope element. The SOAP envelope must conform to the `http://schemas.xmlsoap.org/soap/envelope` namespace. ■ `MustUnderstand`—SOAP header entry not understood by processing party. ■ `Client`—Message was incorrectly formatted or is missing information. ■ `Server`—Problem with the server that prevented message from being processed.
`faultstring`	Human-readable description of fault.
`faultactor`	URI associated with the actor (SOAP node) that caused the fault. In RPC-style messaging, the actor is the URI of the Web service.
`detail`	Application-specific information, such as the exception that was thrown. This element can be an XML structure or plain text.

The following provides an example of a SOAP 1.1 fault message.

Example 16–2 Example of SOAP 1.1 Fault Message

```
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap='http://schemas.xmlsoap.org/soap/envelope'>
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:VersionMismatch</faultcode>
         <faultstring, xml:lang='en">
            Message was not SOAP 1.1 compliant
         </faultstring>
         <faultactor>
            http://sample.org.ocm/jws/authnticator
         </faultactor>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

16.3 Using Modeled Faults

As described previously, a modeled fault is mapped to an exception that is thrown explicitly from the business logic of the Java code. In this case, the exception is mapped to a `wsdl:fault` definitions in the WSDL file, when the Web service is deployed.

The following sections provide more information about using modeled faults:

16.3.1 Creating and Using a Custom Exception

To use modeled faults, you need to create a custom Java exception and throw it from within your Web service.

Example 16–3 provides a simple example of a custom exception being thrown by a a Web service. The exception is called `MissingName` and is thrown when the input argument is empty.

Example 16–3 Web Service With Custom Exception

```
package examples;
import javax.jws.WebService;

@WebService(name="HelloWorld", serviceName="HelloWorldService")
public class HelloWorld {
   public String sayHelloWorld(String message) throws MissingName {
        System.out.println("Say Hello World: " + message);
        if (message == null || message.isEmpty()) {
           throw new MissingName();
        }
      return "Here is the message: '" + message + "'";
      }
}
```

Example 16–4 shows the he class for the exception, `MissingName.java`.

Example 16–4 Custom Exception Class (MissingName)

```
package examples;
import java.lang.Exception;

public class MissingName extends Exception {
    public MissingName() {
        super("Your name is required.");
    }
}
```

16.3.2 How Modeled Faults are Mapped in the WSDL File

The JAX-WS Java-to-WSDL mapping binds subclasses of `java.lang.Exception` to `wsdl:fault` messages. Example 16–4 shows the WSDL that is generated from the annotated Web service in Example 16–3.*

n this example:

- The `<message name="MissingName">` element defines the parts of the `MissingName` message, namely `fault`, and its associated data type, `tns:MissingName`.

```
<message name="MissingName">
  <part name="fault" element="tns:MissingName" />
</message>
```

- The `MissingName` SOAP fault is mapped to the `sayHelloWorld` operation.

```
<operation name="sayHelloWorld">
  <input message="tns:sayHelloWorld" />
  <output message="tns:sayHelloWorldResponse" />
  <fault message="tns:MissingName" name="MissingName" />
</operation>
```

  This `<fault>` subelement in this example is derived from the `throws MissingName` clause of the `sayHelloWorld()` method declaration (see Example 16–3).

```
public String sayHelloWorld(String message) throws MissingName {
...
}
```

- The fault message is mapped to the `sayHelloWorld` operation in the `<binding>` element, as well.

```
<fault name="MissingName">
  <soap:fault name="MissingName" use="literal" />
</fault>
```

Example 16–5 Example of WSDL with Modeled Exceptions

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://examples/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   targetNamespace="http://examples/"
   name="HelloWorldService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://examples/"
     schemaLocation="http://localhost:7001/HelloWorld/HelloWorldService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="sayHelloWorld">
    <part name="parameters" element="tns:sayHelloWorld" />
  </message>
  <message name="sayHelloWorldResponse">
    <part name="parameters" element="tns:sayHelloWorldResponse" />
  </message>
  <message name="MissingName">
    <part name="fault" element="tns:MissingName" />
  </message>
  <portType name="HelloWorld">
    <operation name="sayHelloWorld">
      <input message="tns:sayHelloWorld" />
      <output message="tns:sayHelloWorldResponse" />
      <fault message="tns:MissingName" name="MissingName" />
    </operation>
  </portType>
  <binding name="HelloWorldPortBinding" type="tns:HelloWorld">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document" />
    <operation name="sayHelloWorld">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
```

```
      </input>
      <output>
        <soap:body use="literal" />
       </output>
      <fault name="MissingName">
        <soap:fault name="MissingName" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="HelloWorldService">
    <port name="HelloWorldPort" binding="tns:HelloWorldPortBinding">
      <soap:address
        location="http://localhost:7001/HelloWorld/HelloWorldService" />
    </port>
  </service>
```

16.3.3 How the Fault is Communicated in the SOAP Message

Example 16–6 shows how the SOAP fault is communicated in the resulting SOAP message when the `MissingName` Java exception is thrown.

Example 16–6 Example SOAP Fault Message for MissingName Exception

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Your name is required.</faultstring>
         <detail>
            <ns2:MissingName xmlns:ns2="http://examples/">
               <message>Your name is required.</message>
            </ns2:MissingName>
            <ns2:exception xmlns:ns2="http://jax-ws.dev.java.net/"
 class="examples.MissingName" note="To disable this feature, set
 com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace system
 property to false">
               <message>Your name is required.</message>
               <ns2:stackTrace>
                  <ns2:frame class="examples.HelloWorld" file="HelloWorld.java"
  line="14" method="sayHelloWorld"/>
  ...
               </ns2:stackTrace>
            </ns2:exception>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

16.3.4 Creating the Web Service Client

When you generate a Web service client from a WSDL file that contains mapped faults using `clientgen`, the required exception classes are generated automatically, including the mapped exception, fault bean, service implementation classes client implementation class, which you must modify, as described in the following sections.

- Section 16.3.4.1, "Reviewing the Generated Java Exception Class"
- Section 16.3.4.2, "Reviewing the Generated Java Fault Bean Class"

- Section 16.3.4.3, "Reviewing the Client-side Service Implementation"
- Section 16.3.4.4, "Creating the Client Implementation Class"

For more information about clientgen, see "clientgen" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

16.3.4.1 Reviewing the Generated Java Exception Class

An example of the generated Java exception class is shown in Example 16–7. The `@WebFault` annotation identifies the class as a mapped exception.

Example 16–7 Example of Generated Java Exception Class

```
package examples.client;

import javax.xml.ws.WebFault;

@WebFault(name = "MissingName", targetNamespace = "http://examples/")
public class MissingName_Exception extends Exception {
  private MissingName faultInfo;
  public MissingName_Exception(String message, MissingName faultInfo) { ... }
  public MissingName_Exception(String message, MissingName faultInfo,
     Throwable cause) { ... }
  public MissingName getFaultInfo() { ... }
}
```

16.3.4.2 Reviewing the Generated Java Fault Bean Class

An example of the generated Java fault bean class is shown in Example 16–8, defining the getters and setters for the fault message.

Example 16–8 Example of Generated Java Fault Bean Class

```
package examples.client;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "MissingName", propOrder = {
    "message"
})
public class MissingName {

    protected String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String value) {
        this.message = value;
    }
}
```

16.3.4.3 Reviewing the Client-side Service Implementation

An example of the generated client-side service implementation class is shown in Example 16–9.

Example 16–9 Client-side Service Implementation

```
package examples.client;
...
@WebService(name = "HelloWorld", targetNamespace = "http://examples/")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface HelloWorld {

    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "sayHelloWorld",
       targetNamespace = "http://examples/",
       className = "examples.client.SayHelloWorld")
    @ResponseWrapper(localName = "sayHelloWorldResponse",
       targetNamespace = "http://examples/",
       className = "examples.client.SayHelloWorldResponse")
    public String sayHelloWorld(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0)
        throws MissingName_Exception;
}
```

16.3.4.4 Creating the Client Implementation Class

Create the client implementation class to call the Web service method and throw the custom exception. Then, compile and run the client. For more information about creating Web service clients, see "Invoking Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Example 16–10 shows an example client implementation class.

Example 16–10 Client Implementation Class

```
package examples.client;

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;
import examples.client.MissingName_Exception;

public class Main {

  public static void main(String[] args) throws MissingName_Exception {
    HelloWorldService service;

    try {
      service = new HelloWorldService(new URL(args[0] + "?WSDL"),
                    new QName("http://examples/", "HelloWorldService") );
    } catch (MalformedURLException murl) { throw new RuntimeException(murl); }
      HelloWorld port = service.getHelloWorldPort();

      String result = null;
    try {
```

```
      result = port.sayHelloWorld("");
    } catch (MissingName_Exception e) {
      System.err.println("Error: " + e);
    }
      System.out.println( "Got result: " + result );
  }
}
```

16.4 Using Unmodeled Faults

As noted previously, an unmodeled fault maps to an exception (for example, `java.lang.RuntimeException`) that is generated at run-time when no business logic fault is defined in the WSDL. In this case, Java exceptions are represented as generic SOAP fault exceptions, `javax.xml.ws.soap.SOAPFaultException`.

The following shows an example of an exception that maps to an unmodeled fault.

Example 16–11 Example of Web Service Using Unmodeled Fault

```
package examples;

import javax.jws.WebService;
@WebService(name="HelloWorld", serviceName="HelloWorldService")
public class HelloWorld {
   public String sayHelloWorld(String message) throws MissingName {
        System.out.println("Say Hello World: " + message);
        if (message == null || message.isEmpty()) {
           throw new MissingName(); // Modeled fault
        } else if (message.equalsIgnoreCase("abc")) {
         throw new RuntimeException("Please enter a name."); //Unmodeled fault
        }

      return "Here is the message: '" + message + "'";
   }
}
```

In this example, if the string "abc" is passed to the method, the following `SOAPFaultException` and `RuntimeException` messages are returned in the log file:

Example 16–12 Example of Log File Message for Unmodeled Fault

```
...
run:
  [java] Exception in thread "main" javax.xml.ws.soap.SOAPFaultException: Please
enter a name.
...
Caused by: java.lang.RuntimeException: Please enter a name.\
...
```

16.5 Customizing the Exception Handling Process

You can customize the SOAP fault handling process using *SOAP message handlers*. A SOAP message handler provides a mechanism for intercepting the SOAP message in both the request and response of the Web service. You can create SOAP message handlers to enable Web services and clients to perform additional processing on the SOAP message. For more information, see Chapter 17, "Creating and Using SOAP Message Handlers."

16.6 Disabling the Stack Trace from the SOAP Fault

> **Note:** The `com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace` property is supported as an extension to the JDK 6.0. Because this API is not provided as part of the JDK 6.0 kit, it is subject to change.

By default, the entire stack trace, including nested exceptions, is included in the details of the SOAP fault message. For example, the following shows an example of a SOAP fault message that includes the stack trace:

Example 16–13 Example of Stack Trace in SOAP Fault Message

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope">
   <S:Body>
      <S:Fault xmlns:ns4="http://schemas.xmlsoap.org/soap/envelope/">
         <S:Code>
            <S:Value>S:Receiver</S:Value>
         </S:Code>
         <S:Reason>
            <S:Text xml:lang="en">String index out of range: 3</S:Text>
         </S:Reason>
         <S:Detail>
            <ns2:exception xmlns:ns2="http://jax-ws.dev.java.net/"
             class="java.lang.StringIndexOutOfBoundsException" note="To disable this feature, set
             com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace system property
             to false">
               <message>String index out of range: 3</message>
               <ns2:stackTrace>
                  <ns2:frame class="java.lang.String" file="String.java" line="1934"
                    method="substring"/>
                  <ns2:frame class="ratingservice.CreditRating" file="CreditRating.java"
                    line="21" method="processRating"/>
                  <ns2:frame class="sun.reflect.NativeMethodAccessorImpl"
                    file="NativeMethodAccessorImpl.java" line="native" method="invoke0"/>
                  <ns2:frame class="sun.reflect.NativeMethodAccessorImpl"
                    file="NativeMethodAccessorImpl.java" line="39" method="invoke"/>
                  <ns2:frame class="sun.reflect.DelegatingMethodAccessorImpl"
                    file="DelegatingMethodAccessorImpl.java" line="25" method="invoke"/>
                  <ns2:frame class="java.lang.reflect.Method" file="Method.java" line="597"
                    method="invoke"/>
                  ...
               </ns2:stackTrace>
            </ns2:exception>
         </S:Detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

You can disable the inclusion of the stack trace in the SOAP fault message by setting the `com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace` Java startup property to `false`.

To disable the stack trace:

1. Locate the following entry in the *WL_HOME*`user_projects/domains`*domainName*`/startWebLogic.cmd` file, where *WL_HOME* refers to the main WebLogic Server installation directory:

   ```
   set JAVA_OPTIONS=%SAVE_JAVA_OPTIONS%
   ```

2. Edit the entry as follows:

   ```
   set JAVA_
   OPTIONS=-Dcom.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace=false
   %SAVE_JAVA_OPTIONS%
   ```

3. Save the `startWebLogic.cmd` file.

16.7 Other Exceptions

Note that in addition to the custom exceptions that are thrown explicitly in your Web service and the `SOAPFaultExceptions` that are used to map exceptions that are not caught by your business logic, there are two other exceptions that might be communicated to the Web service client, and that you should be aware of.

Table 16–3 Other Exceptions

Exception	Description
`javax.xml.ws.WebServiceException`	Base exception for all JAX-WS API runtime exceptions, used when calls to JAX-WS Java classes fail, such as `Service.BindingProvider` and `Dispatch`.
`java.util.concurrent.ExecutionException`	Used by JAX-WS asynchronous calls, when a client tries to get the response from an asynchronous call.

17

Creating and Using SOAP Message Handlers

This chapter describes how to create and use SOAP message handlers for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

17.1 Overview of SOAP Message Handlers

Web services and their clients may need to access the SOAP message for additional processing of the message request or response. A SOAP message handler provides a mechanism for intercepting the SOAP message in both the request and response of the Web service. You can create SOAP message handlers to enable Web services and clients to perform additional processing on the SOAP message.

A simple example of using handlers is to access information in the header part of the SOAP message. You can use the SOAP header to store Web service specific information and then use handlers to manipulate it.

You can also use SOAP message handlers to improve the performance of your Web service. After your Web service has been deployed for a while, you might discover that many consumers invoke it with the same parameters. You could improve the performance of your Web service by caching the results of popular invokes of the Web service (assuming the results are static) and immediately returning these results when appropriate, without ever invoking the back-end components that implement the Web service. You implement this performance improvement by using handlers to check the request SOAP message to see if it contains the popular parameters.

JAX-WS supports two types of SOAP message handlers: SOAP handlers and logical handlers. SOAP handlers can access the entire SOAP message, including the message headers and body. Logical handlers can access the payload of the message only, and cannot change any protocol-specific information (like headers) in a message.

Note: If SOAP handlers are used in conjunction with policies (security, WS-ReliableMessaging, MTOM, and so on), for inbound messages, the policy interceptors are executed before the user-defined message handlers. For outbound messages, this order is reversed.

17.2 Adding Server-side SOAP Message Handlers: Main Steps

The following procedure describes the high-level steps to add SOAP message handlers to your Web service.

It is assumed that you have created a basic JWS file that implements a Web service and that you want to update the Web service by adding SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `jwsc` Ant task. For more information, see in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*:

- Use Cases and Examples
- Developing WebLogic Web Services
- Programming the JWS File
- Invoking Web Services

Table 17–1 Steps to Add SOAP Message Handlers to a Web Service

#	Step	Description
1	Design the handlers and handler chains.	Design SOAP message handlers and group them together in a *handler chain*. See Section 17.4, "Designing the SOAP Message Handlers and Handler Chains".
2	For each handler in the handler chain, create a Java class that implements the SOAP message handler interface.	See Section 17.5, "Creating the SOAP Message Handler".
3	Update your JWS file, adding annotations to configure the SOAP message handlers.	See Section 17.6, "Configuring Handler Chains in the JWS File".
4	Create the handler chain configuration file.	See Section 17.7, "Creating the Handler Chain Configuration File".
5	Compile all handler classes in the handler chain and rebuild your Web service.	See Section 17.8, "Compiling and Rebuilding the Web Service".

17.3 Adding Client-side SOAP Message Handlers: Main Steps

You can configure client-side SOAP message handlers for both stand-alone clients and clients that run inside of WebLogic Server. You create the actual Java client-side handler in the same way you create a server-side handler (by creating a Java class that implements the SOAP message handler interface). In many cases you can use the exact same handler class on both the Web service running on WebLogic Server *and* the client applications that invoke the Web service. For example, you can write a generic logging handler class that logs all sent and received SOAP messages, both for the server and for the client.

The following procedure describes the high-level steps to add client-side SOAP message handlers to the client application that invokes a Web service operation.

It is assumed that you have created the client application that invokes a deployed Web service, and that you want to update the client application by adding client-side SOAP message handlers and handler chains. It is also assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file that includes a target for running the `clientgen` Ant task. For more information, see "Invoking a Web service from a Stand-alone Client: Main Steps" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 17–2 Steps to Add SOAP Message Handlers to a Web Service Client

#	Step	Description
1	Design the handlers and handler chains.	This step is similar to designing the server-side SOAP message handlers, except the perspective is from the client application, rather than a Web service. See Section 17.4, "Designing the SOAP Message Handlers and Handler Chains".
2	For each handler in the handler chain, create a Java class that implements the SOAP message handler interface.	This step is similar to designing the server-side SOAP message handlers, except the perspective is from the client application, rather than a Web service. See Section 17.5, "Creating the SOAP Message Handler" for details about programming a handler class.
3	Update your client to programmatically configure the SOAP message handlers.	See Section 17.9, "Configuring the Client-side SOAP Message Handlers".
4	Update the build.xml file that builds your application, specifying to the clientgen Ant task the customization file.	See Section 17.8, "Compiling and Rebuilding the Web Service".
5	Rebuild your client application by running the relevant task.	`prompt> ant build-client`

When you next run the client application, the SOAP messaging handlers listed in the configuration file automatically execute before the SOAP request message is sent and after the response is received.

> **Note:** You do *not* have to update your actual client application to invoke the client-side SOAP message handlers; as long as you specify to the `clientgen` Ant task the handler configuration file, the generated interface automatically takes care of executing the handlers in the correct sequence.

17.4 Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers, you must decide:

- The number of handlers needed to perform the work.
- The sequence of execution.

You group SOAP message handlers together in a *handler chain*. Each handler in a handler chain may define methods for both inbound and outbound messages.

Typically, each SOAP message handler defines a separate set of steps to process the request and response SOAP message because the same type of processing typically must happen for the inbound and outbound message. You can, however, design a handler that processes only the SOAP request and does no equivalent processing of the response. You can also choose not to invoke the next handler in the handler chain and send an immediate response to the client application at any point.

17.4.1 Server-side Handler Execution

When invoking a Web service, WebLogic Server executes handlers as follows:

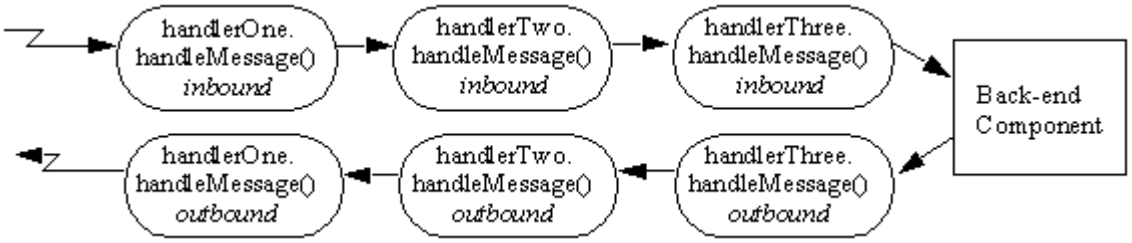
1. The *inbound* methods for handlers in the handler chain are all executed in the order specified by the JWS annotation. Any of these inbound methods might change the SOAP message request.
2. When the last handler in the handler chain executes, WebLogic Server invokes the back-end component that implements the Web service, passing it the final SOAP message request.
3. When the back-end component has finished executing, the *outbound* methods of the handlers in the handler chain are executed in the *reverse* order specified by the JWS annotation. Any of these outbound methods might change the SOAP message response.
4. When the first handler in the handler chain executes, WebLogic Server returns the final SOAP message response to the client application that invoked the Web service.

For example, assume that you are going to use the `@HandlerChain` JWS annotation in your JWS file to specify an external configuration file, and the configuration file defines a handler chain called `SimpleChain` that contains three handlers, as shown in the following sample:

```
<?xml version="1.0" encoding="UTF-8" ?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
   <handler-chain>
      <handler>
         <handler-class>
            Handler1
         </handler-class>
      </handler>
   </handler-chain>
   <handler-chain>
      <handler>
         <handler-class>
            Handler2
         </handler-class>
      </handler>
   </handler-chain>
   <handler-chain>
      <handler>
         <handler-class>
            Handler3
         </handler-class>
      </handler>
   </handler-chain>
</handler-chains>
```

The following graphic shows the order in which WebLogic Server executes the inbound and outbound methods of each handler.

Figure 17–1 Order of Execution of Handler Methods



17.4.2 Client-side Handler Execution

In the case of a client-side handler, the handler executes twice:

- Directly before the client application sends the SOAP request to the Web service
- Directly after the client application receives the SOAP response from the Web service

17.5 Creating the SOAP Message Handler

There are two types of SOAP message handlers that you can create, as defined in the following table.

Table 17–3 Types of SOAP Message Handlers

Handler Type	Description
SOAP handler	Enables you to access the full SOAP message including headers. SOAP handlers are defined using the `javax.xml.ws.handler.soap.SOAPHandler` interface. They are invoked using the import `javax.xml.ws.handler.soap.SOAPMessageContext` which extends `javax.xml.ws.handler.MessageContext` The `SOAPMessageContext.getMessage()` method returns a `javax.xml.soap.SOAPMessage`.
Logical handlers	Provides access to the payload of the message. Logical handlers cannot change any protocol-specific information (like headers) in a message. Logical handlers are defined using the `javax.xml.ws.handler.LogicalHandler` interface (see `http://download.oracle.com/javaee/5/api/javax/xml/ws/handler/LogicalHandler.html`). They are invoked using the `javax.xml.ws.handler.LogicalMessageContext` which extends `javax.xml.ws.handler.MessageContext` The `LogicalMessageContext.getMessage()` method returns a `javax.xml.ws.LogicalMessage`. The payload can be accessed either as a JAXB object or as a `javax.xml.transform.Source` object (see `http://download.oracle.com/javaee/5/api/javax/xml/ws/LogicalMessage.html`).

Each type of message handler extends the `javax.xml.ws.Handler` interface (see `http://download.oracle.com/javaee/5/api/javax/xml/ws/handler/Handler.html`), which defines the methods defined in the following table.

Table 17–4 Handler Interface Methods

Method	Description
`handleMessage()`	Manages normal processing of inbound and outbound messages. A property in the `MessageContext` object is used to determine if the message is inbound or outbound. See Section 17.5.3, "Implementing the Handler.handleMessage() Method".
`handleFault()`	Manages fault processing of inbound and outbound messages. See Section 17.5.4, "Implementing the Handler.handleFault() Method".
`close()`	Concludes the message exchange and cleans up resources that were accessed during processing. See Section 17.5.5, "Implementing the Handler.close() Method".

In addition, you can use the `@javax.annotation.PostConstruct` and `@javax.annotation.PreDestroy` annotations to identify methods that must be executed after the handler is created and before the handler is destroyed, respectively.

Sometimes you might need to directly view or update the SOAP message from within your handler, in particular when handling attachments, such as image. In this case, use the `javax.xml.soap.SOAPMessage` abstract class, which is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification at `http://java.sun.com/webservices/saaj/docs.html` For details, see Section 17.5.7, "Directly Manipulating the SOAP Request and Response Message Using SAAJ".

17.5.1 Example of a SOAP Handler

The following example illustrates a simple SOAP handler that returns whether the message is inbound or outbound along with the message content.

```
package examples.webservices.handler;

import java.util.Set;
import java.util.Collections;
import javax.xml.namespace.QName;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import javax.xml.soap.SOAPMessage;

public class Handler1 implements SOAPHandler<SOAPMessageContext>
{
  public Set<QName> getHeaders()
  {
    return Collections.emptySet();
  }

  public boolean handleMessage(SOAPMessageContext messageContext)
  {
     Boolean outboundProperty = (Boolean)
         messageContext.get (MessageContext.MESSAGE_OUTBOUND_PROPERTY);

     if (outboundProperty.booleanValue()) {
         System.out.println("\nOutbound message:");
     } else {
         System.out.println("\nInbound message:");
     }
```

```
    System.out.println("** Response: "+messageContext.getMessage().toString());
    return true;
  }

  public boolean handleFault(SOAPMessageContext messageContext)
  {
    return true;
  }

  public void close(MessageContext messageContext)
  {
  }
}
```

17.5.2 Example of a Logical Handler

The following example illustrates a simple logical handler that returns whether the message is inbound or outbound along with the message content.

```
package examples.webservices.handler;

import java.util.Set;
import java.util.Collections;
import javax.xml.namespace.QName;
import javax.xml.ws.handler.LogicalHandler;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.LogicalMessageContext;
import javax.xml.ws.LogicalMessage;
import javax.xml.transform.Source;

public class Handler2 implements LogicalHandler<LogicalMessageContext>
{
  public Set<QName> getHeaders()
  {
    return Collections.emptySet();
  }

  public boolean handleMessage(LogicalMessageContext messageContext)
  {
     Boolean outboundProperty = (Boolean)
         messageContext.get (MessageContext.MESSAGE_OUTBOUND_PROPERTY);
     if (outboundProperty.booleanValue()) {
            System.out.println("\nOutbound message:");
     } else {
            System.out.println("\nInbound message:");
     }

    System.out.println("** Response: "+messageContext.getMessage().toString());
    return true;
  }

  public boolean handleFault(LogicalMessageContext messageContext)
  {
    return true;
  }

  public void close(MessageContext messageContext)
  {
  }
}
```

17.5.3 Implementing the Handler.handleMessage() Method

The `Handler.handleMessage()` method is called to intercept a SOAP message request before and after it is processed by the back-end component. Its signature is:

```
public boolean handleMessage(C context)
  throws java.lang.RuntimeException, java.xml.ws.ProtocolException {}
```

Implement this method to perform such tasks as encrypting/decrypting data in the SOAP message before or after it is processed by the back-end component, and so on.

`C` extends `javax.xml.ws.handler.MessageContext` (see http://download.oracle.com/javaee/5/api/javax/xml/ws/handler/MessageContext.html). The `MessageContext` properties allow the handlers in a handler chain to determine if a message is inbound or outbound and to share processing state. Use the `SOAPMessageContext` or `LogicalMessageContext` sub-interface of `MessageContext` to get or set the contents of the SOAP or logical message, respectively. For more information, see Section 17.5.6, "Using the Message Context Property Values and Methods".

After you code all the processing of the SOAP message, code one of the following scenarios:

- Invoke the next handler on the handler request chain by returning `true`.

  The next handler on the request chain is specified as the next `<handler>` subelement of the `<handler-chain>` element in the configuration file specified by the `@HandlerChain` annotation.

- Block processing of the handler request chain by returning `false`.

  Blocking the handler request chain processing implies that the back-end component does not get executed for this invoke of the Web service. You might want to do this if you have cached the results of certain invokes of the Web service, and the current invoke is on the list.

  Although the handler request chain does not continue processing, WebLogic Server does invoke the handler *response* chain, starting at the current handler.

- Throw the `java.lang.RuntimeException` or `java.xml.ws.ProtocolException` for any handler-specific runtime errors.

  WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server log file, and invokes the `handleFault()` method of this handler.

17.5.4 Implementing the Handler.handleFault() Method

The `Handler.handleFault()` method processes the SOAP faults based on the SOAP message processing model. Its signature is:

```
public boolean handleFault(C context)
  throws java.lang.RuntimeException, java.xml.ws.ProtocolException{}
```

Implement this method to handle processing of any SOAP faults generated by the `handleMessage()` method, as well as faults generated by the back-end component.

`C` extends `javax.xml.ws.handler.MessageContext` (see http://download.oracle.com/javaee/5/api/javax/xml/ws/handler/MessageContext.html). The `MessageContext` properties allow the handlers in a handler chain to determine if a message is inbound or outbound and to share processing state.Use the `LogicalMessageContext` or `SOAPMessageContext`

sub-interface of `MessageContext` to get or set the contents of the logical or SOAP message, respectively. For more information, see Section 17.5.6, "Using the Message Context Property Values and Methods".

After you code all the processing of the SOAP fault, do one of the following:

- Invoke the `handleFault()` method on the next handler in the handler chain by returning `true`.
- Block processing of the handler fault chain by returning `false`.

17.5.5 Implementing the Handler.close() Method

The `Handler.close()` method concludes the message exchange and cleans up resources that were accessed during processing. Its signature is:

```
public boolean close(MessageContext context) {}
```

17.5.6 Using the Message Context Property Values and Methods

The following context objects are passed to the SOAP message handlers.

Table 17–5 Message Context Property Values

Message Context Property Values	Description
`javax.xml.ws.handler.LogicalMessageContext`	Context object for logical handlers.
`javax.xml.ws.handler.soap.SOAPMessageContext`	Context object for SOAP handlers.

Each context object extends `javax.xml.ws.handler.MessageContext`, which enables you to access a set of runtime properties of a SOAP message handler from the client application or Web service, or directly from the `javax.xml.ws.WebServiceContext` from a Web service (see `https://jax-ws.dev.java.net/nonav/jax-ws-20-pfd/api/javax/xml/ws/WebServiceContext.html`).

For example, the `MessageContext.MESSAGE_OUTBOUND_PROPERTY` holds a `Boolean` value that is used to determine the direction of a message. During a request, you can check the value of this property to determine if the message is an inbound or outbound request. The property would be `true` when accessed by a client-side handler or `false` when accessed by a server-side handler.

For more information about the `MessageContext` property values that are available, see "Using the MessageContext Property Values" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

The `LogicalMessageContext` class defines the following method for processing the Logical message. For more information, see the `java.xml.ws.handler.LogicalMessageContext` Javadoc at `http://download.oracle.com/javaee/5/api/javax/xml/ws/handler/LogicalMessageContext.html`.

Table 17–6 LogicalMessageContext Class Method

Method	Description
`getMessage()`	Gets a `javax.xml.ws.LogicalMessage` object that contains the SOAP message.

The SOAPMessageContext class defines the following methods for processing the SOAP message. For more information, see the java.xml.ws.handler.soap.SOAPMessageContext Javadoc at http://download.oracle.com/javaee/5/api/javax/xml/ws/handler/soap/SOAPMessageContext.html.

Note: The SOAP message itself is stored in a javax.xml.soap.SOAPMessage object at http://download.oracle.com/javaee/5/api/javax/xml/soap/SOAPMessage.html. For detailed information on this object, see Section 17.5.7, "Directly Manipulating the SOAP Request and Response Message Using SAAJ".

Table 17–7 SOAPMessageContext Class Methods

Method	Description
getHeaders()	Gets headers that have a particular qualified name from the message in the message context.
getMessage()	Gets a javax.xml.soap.SOAPMessage object that contains the SOAP message.
getRoles()	Gets the SOAP actor roles associated with an execution of the handler chain.
setMessage()	Sets the SOAP message.

17.5.7 Directly Manipulating the SOAP Request and Response Message Using SAAJ

The javax.xml.soap.SOAPMessage abstract class is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification at http://java.sun.com/webservices/saaj/docs.html. You use the class to manipulate request and response SOAP messages when creating SOAP message handlers. This section describes the basic structure of a SOAPMessage object and some of the methods you can use to view and update a SOAP message.

A SOAPMessage object consists of a SOAPPart object (which contains the actual SOAP XML document) and zero or more attachments.

Refer to the SAAJ Javadocs for the full description of the SOAPMessage class.

17.5.7.1 The SOAPPart Object

Note: The setContent and getContent methods of the SOAPPart object support javax.xml.transform.stream.StreamSource content only; the methods do not support javax.xml.transform.dom.DOMSource content.

The SOAPPart object contains the XML SOAP document inside of a SOAPEnvelope object. You use this object to get the actual SOAP headers and body.

The following sample Java code shows how to retrieve the SOAP message from a MessageContext object, provided by the Handler class, and get at its parts:

```
SOAPMessage soapMessage =  messageContext.getMessage();
SOAPPart soapPart = soapMessage.getSOAPPart();
```

```
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
SOAPBody soapBody = soapEnvelope.getBody();
SOAPHeader soapHeader = soapEnvelope.getHeader();
```

17.5.7.2 The AttachmentPart Object

The `javax.xml.soap.AttachmentPart` object (see http://download.oracle.com/javaee/5/api/javax/xml/soap/AttachmentPart.html) contains the optional attachments to the SOAP message. Unlike the rest of a SOAP message, an attachment is not required to be in XML format and can therefore be anything from simple text to an image file.

> **Note:** If you are going to access a `java.awt.Image` attachment from your SOAP message handler, see Section 17.5.7.3, "Manipulating Image Attachments in a SOAP Message Handler" for important information.

Use the following methods of the `SOAPMessage` class to manipulate the attachments. For more information, see the `javax.xml.soap.SOAPMessage` Javadoc at http://download.oracle.com/javaee/5/api/javax/xml/soap/SOAPMessage.html.

Table 17–8 SOAPMessage Class Methods to Manipulate Attachments

Method	Description
`addAttachmentPart()`	Adds an `AttachmentPart` object, after it has been created, to the `SOAPMessage`.
`countAttachments()`	Returns the number of attachments in this SOAP message.
`createAttachmentPart()`	Create an `AttachmentPart` object from another type of `Object`.
`getAttachments()`	Gets all the attachments (as `AttachmentPart` objects) into an `Iterator` object.

17.5.7.3 Manipulating Image Attachments in a SOAP Message Handler

It is assumed in this section that you are creating a SOAP message handler that accesses a `java.awt.Image` attachment and that the `Image` has been sent from a client application that uses the client JAX-WS ports generated by the `clientgen` Ant task.

In the client code generated by the `clientgen` Ant task, a `java.awt.Image` attachment is sent to the invoked WebLogic Web service with a MIME type of `text/xml` rather than `image/gif`, and the image is serialized into a stream of integers that represents the image. In particular, the client code serializes the image using the following format:

- `int width`
- `int height`
- `int[] pixels`

This means that, in your SOAP message handler that manipulates the received Image attachment, you must deserialize this stream of data to then re-create the original image.

17.6 Configuring Handler Chains in the JWS File

The `@javax.jws.HandlerChain` annotation (also called `@HandlerChain` in this chapter for simplicity) enables you to configure a handler chain for a Web service. Use the `file` attribute to specify an external file that contains the configuration of the handler chain you want to associate with the Web service. The configuration includes the list of handlers in the chain, the order in which they execute, the initialization parameters, and so on.

The following JWS file shows an example of using the `@HandlerChain` annotation; the relevant Java code is shown in **bold**:

```
package examples.webservices.handler;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.HandlerChain;
import javax.annotation.Resource;
import javax.xml.ws.WebServiceContext;
@WebService(name = "Handler", targetNamespace = "http://example.org")
@HandlerChain(file="handler-chain.xml")
public class HandlerWS
{
  @Resource
  WebServiceContext ctx;
  @WebMethod()
  public String getProperty(String propertyName)
  {
    return (String) ctx.getMessageContext().get(propertyName);
  }
}
```

Before you use the `@HandlerChain` annotation, you must import it into your JWS file, as shown above.

Use the `file` attribute of the `@HandlerChain` annotation to specify the name of the external file that contains configuration information for the handler chain. The value of this attribute is a URL, which may be relative or absolute. Relative URLs are relative to the location of the JWS file at the time you run the `jwsc` Ant task to compile the file.

> **Note:** It is an error to specify more than one `@HandlerChain` annotation in a single JWS file.

For details about creating the external configuration file, see Section 17.7, "Creating the Handler Chain Configuration File".

For additional detailed information about the standard JWS annotations discussed in this section, see the Web services Metadata for the Java Platform specification at `http://www.jcp.org/en/jsr/detail?id=181`.

17.7 Creating the Handler Chain Configuration File

As described in the previous section, you use the `@HandlerChain` annotation in your JWS file to associate a handler chain with a Web service. You must create the handler chain file that consists of an external configuration file that specifies the list of handlers in the handler chain, the order in which they execute, the initialization parameters, and so on.

Because this file is external to the JWS file, you can configure multiple Web services to use this single configuration file to standardize the handler configuration file for all Web services in your enterprise. Additionally, you can change the configuration of the handler chains without needing to recompile all your Web services.

The configuration file uses XML to list one or more handler chains, as shown in the following simple example:

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
      <handler-class>examples.webservices.handler.Handler1</handler-class>
    </handler>
  </handler-chain>
  <handler-chain>
    <handler>
      <handler-class>examples.webservices.handler.Handler2</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

In the example, the handler chain contains two handlers implemented with the class names specified with the `<handler-class>` element. The two handlers execute in forward order before the relevant Web service operation executes, and in reverse order after the operation executes.

Use the `<init-param>` and `<soap-role>`child elements of the `<handler>` element to specify the handler initialization parameters and SOAP roles implemented by the handler, respectively.

You can include logical and SOAP handlers in the same handler chain. At runtime, the handler chain is re-ordered so that all logical handlers are executed before SOAP handlers for an outbound message, and vice versa for an inbound message.

For the XML Schema that defines the external configuration file, additional information about creating it, and additional examples, see the Web services Metadata for the Java Platform specification at http://www.jcp.org/en/jsr/detail?id=181.

17.8 Compiling and Rebuilding the Web Service

It is assumed in this section that you have a working `build.xml` Ant file that compiles and builds your Web service, and you want to update the build file to include handler chain. See "Developing WebLogic Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server* for information on creating this `build.xml` file.

Follow these guidelines to update your development environment to include message handler compilation and building:

- After you have updated the JWS file with the `@HandlerChain` annotation, you must rerun the `jwsc` Ant task to recompile the JWS file and generate a new Web service. This is true anytime you make a change to an annotation in the JWS file.

  If you used the `@HandlerChain` annotation in your JWS file, reran the `jwsc` Ant task to regenerate the Web service, and subsequently changed only the external configuration file, you do not need to rerun `jwsc` for the second change to take affect.

- The `jwsc` Ant task compiles SOAP message handler Java files into handler classes (and then packages them into the generated application) if all the following conditions are true:
  - The handler classes are referenced in the `@HandlerChain` annotation of the JWS file.
  - The Java files are located in the directory specified by the `sourcepath` attribute.
  - The classes are not currently in your CLASSPATH.

  If you want to compile the handler classes yourself, rather than let `jwsc` compile them automatically, ensure that the compiled classes are in your CLASSPATH before you run the `jwsc` Ant task.
- You deploy and invoke a Web service that has a handler chain associated with it in the same way you deploy and invoke one that has no handler chain. The only difference is that when you invoke any operation of the Web service, the WebLogic Web services runtime executes the handlers in the handler chain both before and after the operation invoke.

17.9 Configuring the Client-side SOAP Message Handlers

You configure client-side SOAP message handlers in one of the following ways:

- Set a handler chain directly on the `javax.xml.ws.BindingProvider`, such as a port proxy or `javax.xml.ws.Dispatch` object. For example:

```
package examples.webservices.handler.client;

import javax.xml.namespace.QName;
import java.net.MalformedURLException;
import java.net.URL;

import javax.xml.ws.handler.Handler;
import javax.xml.ws.Binding;
import javax.xml.ws.BindingProvider;
import java.util.List;

import examples.webservices.handler.Handler1;
import examples.webservices.handler.Handler2;

public class Main {
  public static void main(String[] args) {
     HandlerWS test;
     try {
          test = new HandlerWS(new URL(args[0] + "?WSDL"), new
                  QName("http://example.org", "HandlerWS") );
        } catch (MalformedURLException murl) { throw new
RuntimeException(murl); }
      HandlerWSPortType port = test.getHandlerWSPortTypePort();

      Binding binding = ((BindingProvider)port).getBinding();
      List<Handler> handlerList = binding.getHandlerChain();
      handlerList.add(new Handler1());
      handlerList.add(new Handler2());
      binding.setHandlerChain(handlerList);
      String result = null;
      result = port.sayHello("foo bar");
      System.out.println( "Got result: " + result );
```

```
  }
}
```

- Implement a `javax.xml.ws.handler.HandlerResolver` on a `Service` instance. For example:

```
public static class MyHandlerResolver implements HandlerResolver {
   public List<Handler> getHandlerChain(PortInfo portInfo) {
     List<Handler> handlers = new ArrayList<Handler>();
     // add handlers to list based on PortInfo information
     return handlers;
   }
}
```

  Add a handler resolver to the `Service` instance using the `setHandlerResolver()` method. In this case, the port proxy or `Dispatch` object created from the `Service` instance uses the `HandlerResolver` to determine the handler chain. For example:

```
test.setHandlerResolver(new MyHandlerResolver());
```

- Create a customization file that includes a `<binding>` element that contains a handler chain description. The schema for the `<handler-chains>` element is the same for both handler chain files (on the server) and customization files. For example:

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:7001/handler/HandlerWS?WSDL"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions"
        xmlns:jws="http://java.sun.com/xml/ns/javaee">
    <handler-chains>
       <handler-chain>
          <handler>
             <handler-class>examples.webservices.handler.Handler1
             </handler-class>
          </handler>
       </handler-chain>
       <handler-chain>
          <handler>
             <handler-class>examples.webservices.handler.Handler2
             </handler-class>
          </handler>
       </handler-chain>
    </handler-chains>
</bindings>
```

  Use the `<binding>` child element of the `clientgen` command to pass the customization file.

18

Sending and Receiving SOAP Headers

This chapter describes how use the methods available from `com.sun.xml.ws.developer.WSBindingProvider` to send outbound or receive inbound SOAP headers for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

- Section 18.1, "Overview of Sending and Receiving SOAP Headers"
- Section 18.2, "Sending SOAP Headers Using WSBindingProvider"
- Section 18.3, "Receiving SOAP Headers Using WSBindingProvider"

> **Note:** The `com.sun.xml.ws.developer.WSBindingProvider` and `com.sun.xml.ws.api.message.Headers` APIs are supported as an extension to the JDK 6.0. Because the APIs are not provided as part of the JDK 6.0 kit, they are subject to change.

18.1 Overview of Sending and Receiving SOAP Headers

When you create a proxy or Dispatch client, the client implements the `javax.xml.ws.BindingProvider` interface. If you need to send or receive a SOAP header, you can downcast the Web service proxy or Dispatch client to `com.sun.xml.ws.developer.WSBindingProvider` and use the methods on the interface to send outbound or receive inbound SOAP headers.

18.2 Sending SOAP Headers Using WSBindingProvider

Use the `setOutboundHeaders` method to the `com.sun.xml.ws.developer.WSBindingProvider` to send SOAP headers. You create SOAP headers using the `com.sun.xml.ws.api.message.Headers` method.

For example, the following provides a code excerpt showing how to pass a simple string value as a header.

Example 18–1 Sending SOAP Headers Using WSBindingProvider

```
import com.sun.xml.ws.developer.WSBindingProvider;
import com.sun.xml.ws.api.message.Headers;
import javax.xml.namespace.QName;
...
HelloService helloService = new HelloService();
HelloPort port = helloService.getHelloPort();
WSBindingProvider bp = (WSBindingProvider)port;
```

```
bp.setOutboundHeaders(
  // Sets a simple string value as a header
  Headers.create(new QName("simpleHeader"),"stringValue")
);
...
```

18.3 Receiving SOAP Headers Using WSBindingProvider

Use the `getInboundHeaders` method to the `com.sun.xml.ws.developer.WSBindingProvider` to receive SOAP headers.

For example, the following provides a code excerpt showing how to get inbound headers.

Example 18–2 Receiving SOAP Headers Using WSBindingProvider

```
import com.sun.xml.ws.developer.WSBindingProvider;
import com.sun.xml.ws.api.message.Headers;
import javax.xml.namespace.QName;
import java.util.List;
...
HelloService helloService = new HelloService();
HelloPort port = helloService.getHelloPort();
WSBindingProvider bp = (WSBindingProvider)port;

List inboundHeaders = bp.getInboundHeaders();
...
```

19

Operating at the XML Message Level

This chapter describes how to develop Web service provider-based endpoints and dispatch clients to operate at the XML message level for WebLogic Web services using Java API for XML Web Services (JAX-WS).

This chapter includes the following sections:

19.1 Overview of Web Service Provider-based Endpoints and Dispatch Clients

Although the use of JAXB-generated classes is simpler, faster, and likely to be less error prone, there are times when you may want to generate your own business logic to manipulate the XML message content directly. Message-level access can be accomplished on the server side using Web service Provider-based endpoints, and on the client side using Dispatch clients.

A **Web service Provider-based endpoint** offers a dynamic alternative to the Java service endpoint interface (SEI)-based endpoint. Unlike the SEI-based endpoint that abstracts the details of converting between Java objects and their XML representation, the Provider interface enables you to access the content directly at the XML message level—without the JAXB binding. Web service Provider-based endpoints can be implemented synchronously or asynchronously using the `javax.xml.ws.Provider<T>` or `com.sun.xml.ws.api.server.AsyncProvider<T>` interfaces, respectively. For more information about developing Web service Provider-based endpoints, see Section 19.3, "Developing a Web Service Provider-based Endpoint (Starting from Java)."

A **Web service Dispatch client**, implemented using the `javax.xml.ws.Dispatch<T>` interface, enables clients to work with messages at

the XML level. The steps to develop a Web service Dispatch client are described in Section 19.6, "Developing a Web Service Dispatch Client".

Provider endpoints and Dispatch clients can be used in combination with other WebLogic Web services features as long as a WSDL is available, including:

- WS-Security
- WS-ReliableMessaging
- WS-MakeConnection
- WS-AtomicTransaction

In addition, Dispatch clients can be used in combination with the asynchronous client transport and asynchronous client handler features. These features are described in detail in Chapter 4, "Invoking Web Services Asynchronously," and a code example is provided in Section 19.6.2, "Creating a Dispatch Instance."

19.2 Usage Modes and Message Formats for Operating at the XML Level

When operating on messages at the XML level using Provider-based endpoints or Dispatch clients, you use one of the usage modes defined in the following table. You define the usage mode using the `javax.xml.ws.ServiceMode` annotation, as described in Section 19.3.5, "Specifying the Usage Mode (@ServiceMode Annotation)."

Table 19–1 Usage Modes for Operating at the XML Message Level

Usage Mode	Description
Message	Operates directly with the entire message. For example, if a SOAP binding is used, then the entire SOAP envelope is accessed.
Payload	Operates on the payload of a message only. For example, if a SOAP binding is used, then the SOAP body is accessed.

Provider-based endpoints and Dispatch clients can receive and send messages using one of the message formats defined in Table 19–2. This table also defines the valid message format and usage mode combinations based on the configured binding type (SOAP or XML over HTTP).

Table 19–2 Message Formats Supported for Operating at the XML Message Level

Message Format	Usage Mode Support for SOAP/HTTP Binding	Usage Mode Support for XML/HTTP Binding
`javax.xml.transform.Source`	**Message mode**: SOAP envelope **Payload mode**: SOAP body	**Message mode**: XML content as `Source` **Payload mode**: XML content as `Source`
`javax.activation.DataSource`	Not valid in either mode because attachments in SOAP/HTTP binding are sent using `SOAPMessage` format.	**Message mode**: `DataSource` object Not valid in payload mode because `DataSource` is used for sending attachments.
`javax.xml.soap.SOAPMessage`	**Message mode**: SOAPMessage object Not valid in payload mode because the entire SOAP message is received, not just the payload.	Not valid in either mode because the client can send a non-SOAP message in XML/HTTP binding.

19.3 Developing a Web Service Provider-based Endpoint (Starting from Java)

You can develop both synchronous and asynchronous Web service Provider-based endpoints, as described in the following sections:

- Section 19.3.1, "Developing a Synchronous Provider-based Endpoint"
- Section 19.3.2, "Developing an Asynchronous Provider-based Endpoint"

> **Note:** To start from WSDL and flag a port as a Web service provider, see Section 19.4, "Developing a Web Service Provider-based Endpoint (Starting from WSDL)".

19.3.1 Developing a Synchronous Provider-based Endpoint

A Web service Provider-based endpoint, implemented using the `javax.xml.ws.Provider<T>`, enables you to access content directly at the XML message level—without the JAXB binding. The `Provider` interface processes messages synchronously—the service waits to process the response before continuing with its work. For more information about the `javax.xml.ws.Provider<T>` interface, see `http://download.oracle.com/javaee/5/api/javax/xml/ws/Provider.html`.

The following procedure describes the typical steps for programming a JWS file that implements a synchronous Web service Provider-based endpoint.

Table 19–3 Steps to Develop a Synchronous Web Service Provider-based Endpoint

#	Step	Description

1	Import the JWS annotations that will be used in your Web service Provider-based JWS file.	The standard JWS annotations for a Web service Provider-based JWS file include:
		`import javax.xml.ws.Provider;`
		`import javax.xml.ws.WebServiceProvider;`
		`import javax.xml.ws.ServiceMode;`
		
		Import additional annotations, as required. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

2	Specify one of the message formats supported, defined in Table 19–2, when developing the Provider-based implementation class.	See Section 19.3.3, "Specifying the Message Format".

Table 19–3 (Cont.) Steps to Develop a Synchronous Web Service Provider-based Endpoint

#	Step	Description
3	Add the standard required `@WebServiceProvider` JWS annotation at the class level to specify that the Java class exposes a Web service provider.	See Section 19.3.4, "Specifying that the JWS File Implements a Web Service Provider (@WebServiceProvider Annotation)."
4	Add the standard `@ServiceMode` JWS annotation at the class level to specify whether the Web service provider is accessing information at the message or message payload level. (Optional)	See Section 19.3.5, "Specifying the Usage Mode (@ServiceMode Annotation)." The service mode defaults to `Service.Mode.Payload`.
5	Define the `invoke()` method.	The `invoke()` method is called and provides the message or message payload as input to the method using the specified message format. See Section 19.3.6, "Defining the invoke() Method for a Synchronous Provider-based Endpoints."

The following sample JWS file shows how to implement a simple synchronous Web service Provider-based endpoint. The steps to develop a synchronous Web service Provider-based endpoint are described in detail in the sections that follow. To review the JWS file within the context of a complete sample, see "Creating JAX-WS Web Services for Java EE" in the Web Services Samples distributed with Oracle WebLogic Server.

> **Note:** RESTful Web Services can be built using XML/HTTP binding Provider-based endpoints. For an example of programming a Provider-based endpoint within the context of a RESTful Web service, see Section 20, "Programming Web Services Using XML Over HTTP."

Example 19–1 Example of a JWS File that Implements a Synchronous Provider- based Endpoint

```
package examples.webservices.jaxws;

import org.w3c.dom.Node;

import javax.xml.transform.Source;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Provider;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.Service;
import java.io.ByteArrayInputStream;


/**
 * A simple Provider-based Web service implementation.
 *
 * @author Copyright (c) 2010, Oracle and/or its affiliates.
 * All Rights Reserved.
```

```
 */
// The @ServiceMode annotation specifies whether the Provider instance
// receives entire messages or message payloads.
@ServiceMode(value = Service.Mode.PAYLOAD)

// Standard JWS annotation that configures the Provider-based Web service.
@WebServiceProvider(portName = "SimpleClientPort",
    serviceName = "SimpleClientService",
    targetNamespace = "http://jaxws.webservices.examples/",
    wsdlLocation = "SimpleClientService.wsdl")
public class SimpleClientProviderImpl implements Provider<Source> {

  //Invokes an operation according to the contents of the request message.
  public Source invoke(Source source) {
    try {
      DOMResult dom = new DOMResult();
      Transformer trans = TransformerFactory.newInstance().newTransformer();
      trans.transform(source, dom);
      Node node = dom.getNode();
      // Get the operation name node.
      Node root = node.getFirstChild();
      // Get the parameter node.
      Node first = root.getFirstChild();
      String input = first.getFirstChild().getNodeValue();
      // Get the operation name.
      String op = root.getLocalName();
      if ("invokeNoTransaction".equals(op)) {
        return sendSource(input);
      } else {
        return sendSource2(input);
      }
    }
    catch (Exception e) {
      throw new RuntimeException("Error in provider endpoint", e);
    }
  }

  private Source sendSource(String input) {
    String body =
        "<ns:invokeNoTransactionResponse
             xmlns:ns=\"http://jaxws.webservices.examples/\"><return>"
           + "constructed:" + input
           + "</return></ns:invokeNoTransactionResponse>";
    Source source = new StreamSource(new ByteArrayInputStream(body.getBytes()));
    return source;
  }

  private Source sendSource2(String input) {
    String body =
        "<ns:invokeTransactionResponse
            xmlns:ns=\"http://jaxws.webservices.examples/\"><return>"
           + "constructed:" + input
           + "</return></ns:invokeTransactionResponse>";
    Source source = new StreamSource(new ByteArrayInputStream(body.getBytes()));
    return source;
  }

}
```

19.3.2 Developing an Asynchronous Provider-based Endpoint

As with the `Provider` interface, Web service Provider-based endpoints implemented using the `com.sun.xml.ws.api.server.AsyncProvider<T>` interface enable you to access content directly at the XML message level—without the JAXB binding. However, the `AsyncProvider` interface processes messages asynchronously—the service can continue its work and process the request when it becomes available, without blocking the thread.

The following procedure describes the typical steps for programming a JWS file that implements an asynchronous Web service Provider-based endpoint.

Table 19–4 Steps to Develop an Asynchronous Web Service Provider-based Endpoint

#	Step	Description

1	Import the JWS annotations that will be used in your Web service Provider-based JWS file.	The standard JWS annotations for an asynchronous Web service Provider-based JWS file include:
		`import com.sun.xml.ws.api.server.AsyncProvider;`
		`import com.sun.xml.ws.api.server.AsyncProviderCallback;`
		`import javax.xml.ws.ServiceMode;`
		Import additional annotations, as required. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

2	Specify one of the message formats supported, defined in Table 19–2, when developing the Provider-based implementation class.	See Section 19.3.3, "Specifying the Message Format".
3	Add the standard required `@WebServiceProvider` JWS annotation at the class level to specify that the Java class exposes a Web service provider.	See Section 19.3.4, "Specifying that the JWS File Implements a Web Service Provider (@WebServiceProvider Annotation)."
4	Add the standard `@ServiceMode` JWS annotation at the class level to specify whether the Web service provider is accessing information at the message or message payload level. (Optional)	See Section 19.3.5, "Specifying the Usage Mode (@ServiceMode Annotation)." The service mode defaults to `Service.Mode.Payload`.
5	Define the `invoke()` method.	The `invoke()` method is called and provides the message or message payload as input to the method using the specified message format. See Section 19.3.7, "Defining the invoke() Method for an Asynchronous Provider-based Endpoints."
6	Define the asynchronous handler callback method to handle the response.	The method handles the response when it is returned. See Section 19.3.8, "Defining the Callback Handler for the Asynchronous Provider-based Endpoint."

The following sample JWS file shows how to implement a simple asynchronous Web service Provider-based endpoint. The steps to develop an asynchronous Web service Provider-based endpoint are described in detail in the sections that follow.

Example 19–2 Example of a JWS File that Implements an Asynchronous Provider-based Endpoint

```
package asyncprovider.server;
```

```
import com.sun.xml.ws.api.server.AsyncProvider;
import com.sun.xml.ws.api.server.AsyncProviderCallback;

import javax.xml.bind.JAXBContext;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.WebServiceProvider;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

@WebServiceProvider(
    wsdlLocation="WEB-INF/wsdl/hello_literal.wsdl",
    targetNamespace = "urn:test",
    serviceName="Hello")

public class HelloAsyncImpl implements AsyncProvider<Source> {

    private static final JAXBContext jaxbContext = createJAXBContext();
    private int bodyIndex;

    public javax.xml.bind.JAXBContext getJAXBContext(){
        return jaxbContext;
    }

    private static javax.xml.bind.JAXBContext createJAXBContext(){
        try{
            return javax.xml.bind.JAXBContext.newInstance(ObjectFactory.class);
        }catch(javax.xml.bind.JAXBException e){
            throw new WebServiceException(e.getMessage(), e);
        }
    }

    private Source sendSource() {
        System.out.println("**** sendSource ******");

        String[] body  = {
            "<HelloResponse xmlns=\"urn:test:types\">
              <argument xmlns=\"\">foo</argument>
              <extra xmlns=\"\">bar</extra>
             </HelloResponse>",
            "<ans1:HelloResponse xmlns:ans1=\"urn:test:types\">
               <argument>foo</argument>
               <extra>bar</extra>
             </ans1:HelloResponse>",
        };
        int i = (++bodyIndex)%body.length;
        return new StreamSource(
            new ByteArrayInputStream(body[i].getBytes()));
    }

    private Hello_Type recvBean(Source source) throws Exception {
        System.out.println("**** recvBean ******");
        return (Hello_Type)jaxbContext.createUnmarshaller().unmarshal(source);
    }

    private Source sendBean() throws Exception {
        System.out.println("**** sendBean ******");
        HelloResponse resp = new HelloResponse();
```

```
        resp.setArgument("foo");
        resp.setExtra("bar");
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        jaxbContext.createMarshaller().marshal(resp, bout);
        return new StreamSource(new ByteArrayInputStream(bout.toByteArray()));
    }

    public void invoke(Source source, AsyncProviderCallback<Source> cbak,
                       WebServiceContext ctxt) {
        System.out.println("**** Received in AsyncProvider Impl ******");
        try {
             Hello_Type hello = recvBean(source);
             String arg = hello.getArgument();
             if (arg.equals("sync")) {
                 String extra = hello.getExtra();
                 if (extra.equals("source")) {
                        cbak.send(sendSource());
                 } else if (extra.equals("bean")) {
                        cbak.send(sendBean());
                 } else {
                        throw new WebServiceException("Expected extra =
                             (source|bean|fault), Got="+extra);
                 }
             } else if (arg.equals("async")) {
               new Thread(new RequestHandler(cbak, hello)).start();
             } else {
                throw new WebServiceException("Expected Argument =
                  (sync|async), Got="+arg);
             }
        } catch(Exception e) {
          throw new WebServiceException("Endpoint failed", e);
        }
    }

    private class RequestHandler implements Runnable {
        final AsyncProviderCallback<Source> cbak;
        final Hello_Type hello;
        public RequestHandler(AsyncProviderCallback<Source> cbak, Hello_Type hello) {
            this.cbak = cbak;
            this.hello = hello;
        }

        public void run() {
            try {
                Thread.sleep(5000);
            } catch(InterruptedException ie) {
                  cbak.sendError(new WebServiceException("Interrupted..."));
                  return;
            }
            try {
                String extra = hello.getExtra();
                if (extra.equals("source")) {
                    cbak.send(sendSource());
                } else if (extra.equals("bean")) {
                    cbak.send(sendBean());
                } else {
                    cbak.sendError(new WebServiceException(
                     "Expected extra = (source|bean|fault), Got="+extra));
                }
           } catch(Exception e) {
```

```
                cbak.sendError(new WebServiceException(e));
            }
        }
    }
}
```

19.3.3 Specifying the Message Format

Specify one of the message formats supported, defined in Table 19–2, when developing the Provider-based implementation class.

For example, in the Provider implementation example shown in Example 19–1, "Example of a JWS File that Implements a Synchronous Provider- based Endpoint", the `SimpleClientProviderImpl` class implements the `Provider<Source>` interface, indicating that both the input and output types are `java.xml.transform.Source` objects.

```
public class SimpleClientProviderImpl implements Provider<Source> {
. . .
}
```

Similarly, in the AsyncProvider implementation example shown in Example 19–2, "Example of a JWS File that Implements an Asynchronous Provider-based Endpoint", the `HelloAsyncImpl` class implements the `AsyncProvider<Source>` interface, indicating that both the input and output types are `java.xml.transform.Source` objects.

```
public class HelloAsyncImpl implements AsyncProvider<Source>  {
. . .
}
```

19.3.4 Specifying that the JWS File Implements a Web Service Provider (@WebServiceProvider Annotation)

Use the standard `javax.xml.ws.WebServiceProvider` annotation to specify, at the class level, that the JWS file implements a Web service provider, as shown in the following code excerpt:

```
@WebServiceProvider(portName = "SimpleClientPort",
    serviceName = "SimpleClientService",
    targetNamespace = "http://jaxws.webservices.examples/",
    wsdlLocation = "SimpleClientService.wsdl")
```

In the example, the service name is `SimpleClientService`, which will map to the `wsdl:service` element in the generated WSDL file. The port name is `SimpleClientPort`, which will map to the `wsdl:port` element in the generated WSDL. The target namespace used in the generated WSDL is `http://jaxws.webservices.examples/` and the WSDL location is local to the Web service provider, at `SimpleClientService.wsdl`.

For more information about the `@WebServiceProvider` annotation, see `https://jax-ws.dev.java.net/nonav/2.1.5/docs/annotations.html`.

19.3.5 Specifying the Usage Mode (@ServiceMode Annotation)

The `javax.xml.ws.ServiceMode` annotation is used to specify whether the Web service Provider-based endpoint receives entire messages (`Service.Mode.MESSAGE`) or message payloads (`Service.Mode.PAYLOAD`) only.

For example:

```
@ServiceMode(value = Service.Mode.PAYLOAD)
```

If not specified, the `@ServiceMode` annotation defaults to `Service.Mode.PAYLOAD`.

For a list of valid message format and usage mode combinations, see Table 19–2.

For more information about the `@ServiceMode` annotation, see https://jax-ws.dev.java.net/nonav/2.1.4/docs/annotations.html.

19.3.6 Defining the invoke() Method for a Synchronous Provider-based Endpoints

The `Provider<T>` interface defines a single method that you must define in your implementation class:

```
T invoke(T request)
```

When a Web service request is received, the `invoke()` method is called and provides the message or message payload as input to the method using the specified message format.

For example, in the Provider implementation example shown in Section 19–1, "Example of a JWS File that Implements a Synchronous Provider- based Endpoint", excerpted below, the class defines an `invoke` method to take as input the `Source` parameter and return a `Source` response.

```
public Source invoke(Source source) {
  try {
    DOMResult dom = new DOMResult();
    Transformer trans = TransformerFactory.newInstance().newTransformer();
    trans.transform(source, dom);
    Node node = dom.getNode();
    // Get the operation name node.
    Node root = node.getFirstChild();
    // Get the parameter node.
    Node first = root.getFirstChild();
    String input = first.getFirstChild().getNodeValue();
    // Get the operation name.
    String op = root.getLocalName();
    if ("invokeNoTransaction".equals(op)) {
      return sendSource(input);
    } else {
      return sendSource2(input);
    }
  }
  catch (Exception e) {
    throw new RuntimeException("Error in provider endpoint", e);
  }
}
```

19.3.7 Defining the invoke() Method for an Asynchronous Provider-based Endpoints

The `AsycnProvider<T>` interface defines a single method that you must define in your implementation class:

```
void invoke(T request, AsyncProviderCallback<t> callback, WebserviceContext
context))
```

You pass the following parameters to the invoke method:

- Request message or message payload in the specified format.
- `com.sun.xml.ws.api.server.AsyncProviderCallback` implementation that will handle the response once it is returned. For more information, see Section 19.3.8, "Defining the Callback Handler for the Asynchronous Provider-based Endpoint.".
- The `javax.xml.ws.WebServiceContext` that defines the message context for the request being served. An asynchronous Provider-based endpoint cannot use the injected `WebServiceContext` which relies on the calling thread to determine the request it should return information about. Instead, it passes the `WebServiceContext` object which remains usable until you invoke `AsyncProviderCallback`.

For example, in the AysncProvider implementation example shown in Example 19–2, "Example of a JWS File that Implements an Asynchronous Provider-based Endpoint", excerpted below, the class defines an `invoke` method as shown below:

```
public void invoke(Source source, AsyncProviderCallback<Source> cbak,
                   WebServiceContext ctxt) {
    System.out.println("**** Received in AsyncProvider Impl ******");
    try {
         Hello_Type hello = recvBean(source);
         String arg = hello.getArgument();
         if (arg.equals("sync")) {
             String extra = hello.getExtra();
             if (extra.equals("source")) {
                    cbak.send(sendSource());
             } else if (extra.equals("bean")) {
                    cbak.send(sendBean());
             } else {
                    throw new WebServiceException("Expected extra =
                         (source|bean|fault), Got="+extra);
             }
         } else if (arg.equals("async")) {
           new Thread(new RequestHandler(cbak, hello)).start();
         } else {
            throw new WebServiceException("Expected Argument =
              (sync|async), Got="+arg);
         }
    } catch(Exception e) {
      throw new WebServiceException("Endpoint failed", e);
    }
}
```

19.3.8 Defining the Callback Handler for the Asynchronous Provider-based Endpoint

The `AsyncProviderCallback` interface enables you to define a callback handler for processing the asynchronous response once it is received.

For example, in the AysncProvider implementation example shown in Example 19–2, "Example of a JWS File that Implements an Asynchronous Provider-based Endpoint", excerpted below, the `RequestHandler` method uses the AsyncProviderCallback callback handler to process the asynchronous response.

```
    private class RequestHandler implements Runnable {
        final AsyncProviderCallback<Source> cbak;
        final Hello_Type hello;
        public RequestHandler(AsyncProviderCallback<Source> cbak, Hello_Type
hello) {
            this.cbak = cbak;
            this.hello = hello;
        }

        public void run() {
            try {
                Thread.sleep(5000);
            } catch(InterruptedException ie) {
                  cbak.sendError(new WebServiceException("Interrupted..."));
                  return;
            }
            try {
                String extra = hello.getExtra();
                if (extra.equals("source")) {
                    cbak.send(sendSource());
                } else if (extra.equals("bean")) {
                    cbak.send(sendBean());
                } else {
                    cbak.sendError(new WebServiceException(
                     "Expected extra = (source|bean|fault), Got="+extra));
                }
           } catch(Exception e) {
                cbak.sendError(new WebServiceException(e));
           }
       }
    }
}
```

19.4 Developing a Web Service Provider-based Endpoint (Starting from WSDL)

If the Provider-based endpoint is being generated from a WSDL file, the `<provider>` WSDL extension can be used to mark a port as a provider. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings wsdlLocation="SimpleClientService.wsdl"
  xmlns="http://java.sun.com/xml/ns/jaxws">
<bindings node="wsdl:definitions" >
    <package name="provider.server"/>
   <provider>true</provider>
</bindings>
```

19.5 Using SOAP Handlers with Provider-based Endpoints

Provider-based endpoints may need to access the SOAP message for additional processing of the message request or response. You can create SOAP message handlers to enable Provider-based endpoints to perform this additional processing on the SOAP message, just as you do for an SEI-based endpoint. For more information about creating the SOAP handler, see "Creating the SOAP Message Handler" on page 17-5.

Table 17–1, " Steps to Add SOAP Message Handlers to a Web Service" enumerates the steps required to add a SOAP handler to a Web service. These steps apply to Web service Provider-based endpoints, as well.

For example:

1. Design SOAP message handlers and group them together in a *handler chain*, as described in Section 17.4, "Designing the SOAP Message Handlers and Handler Chains.".
2. For each handler in the handler chain, create a Java class that implements the SOAP message handler interface, as described in Section 17.5, "Creating the SOAP Message Handler."

   An example of the SOAP handler, `MyHandler`, is shown below.

```
package provider.rootpart_charset_772.server;

import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.xml.namespace.QName;
import javax.xml.soap.AttachmentPart;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import java.io.InputStream;
import java.io.OutputStream;.import java.io.ByteArrayInputStream;
import java.util.Set;

public class MyHandler implements SOAPHandler<SOAPMessageContext> {

    public Set<QName> getHeaders() {
        return null;
    }

    public boolean handleMessage(SOAPMessageContext smc) {
      if (!(Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY))
            return true;
        try {
            SOAPMessage msg = smc.getMessage();
        AttachmentPart part = msg.createAttachmentPart(getDataHandler());
          part.setContentId("SOAPTestHandler@example.jaxws.sun.com");
            msg.addAttachmentPart(part);
            msg.saveChanges();
            smc.setMessage(msg);
        } catch (Exception e) {
            throw new WebServiceException(e);
        }
        return true;
    }

    public boolean handleFault(SOAPMessageContext context) {
        return true;
    }

    public void close(MessageContext context) {}

    private DataHandler getDataHandler() throws Exception {
        return new DataHandler(new DataSource() {
            public String getContentType() {
```

```
                return "text/xml";
            }

            public InputStream getInputStream() {
                return new ByteArrayInputStream("<a/>".getBytes());
            }

            public String getName() {
                return null;
            }

            public OutputStream getOutputStream() {
                throw new UnsupportedOperationException();
            }
        });
    }
}
```

3. Add the `@javax.jws.HandlerChain` annotation to the Provider implementation, as described in Section 17.6, "Configuring Handler Chains in the JWS File."

   For example:

```
package provider.rootpart_charset_772.server;

import javax.jws.HandlerChain;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.*;
import java.io.ByteArrayInputStream;

@WebServiceProvider(targetNamespace="urn:test", portName="HelloPort",
serviceName="Hello")
@ServiceMode(value=Service.Mode.MESSAGE)
@HandlerChain(file="handlers.xml")
public class SOAPMsgProvider implements Provider<SOAPMessage> {

    public SOAPMessage invoke(SOAPMessage msg) {
        try {
    // keeping white space in the string is intentional
            String content = "<soapenv:Envelope
                xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\">
                <soapenv:Body>  <VoidTestResponse
                xmlns=\"urn:test:types\">
                </VoidTestResponse></soapenv:Body></soapenv:Envelope>";
            Source source = new StreamSource(new
                ByteArrayInputStream(content.getBytes()));
            MessageFactory fact = MessageFactory.newInstance();
            SOAPMessage soap = fact.createMessage();
            soap.getSOAPPart().setContent(source);
            soap.getMimeHeaders().addHeader("foo", "bar");
            return soap;
        } catch(Exception e) {
            throw new WebServiceException(e);
        }
    }
}
```

4. Create the handler chain configuration file, as described in Section 17.7, "Creating the Handler Chain Configuration File."

   An example of the handler chain configuration file, `handlers.xml`, is shown below.

```
<handler-chains xmlns='http://java.sun.com/xml/ns/javaee'>
   <handler-chain>
      <handler>
         <handler-name>MyHandler</handler-name>
         <handler-class>
             provider.rootpart_charset_772.server.MyHandler
         </handler-class>
      </handler>
   </handler-chain>
</handler-chains>
```

5. Compile all handler classes in the handler chain and rebuild your Web service, as described in Section 17.8, "Compiling and Rebuilding the Web Service.".

19.6 Developing a Web Service Dispatch Client

A Web service Dispatch client, implemented using the `javax.xml.ws.Dispatch<T>` interface, enables clients to work with messages at the XML level.

The following procedure describes the typical steps for programming a Web service Dispatch client.

Table 19–5 Steps to Develop a Web Service Provider-based Endpoint

#	Step	Description

1	Import the JWS annotations that will be used in your Web service Provider-based JWS file.	The standard JWS annotations for a Web service Provider-based JWS file include:
		`import javax.xml.ws.Service;`
		`import javax.xml.ws.Dispatch;`
		`import javax.xml.ws.ServiceMode;`
		
		Import additional annotations, as required. For a complete list of JWS annotations that are supported, see "Web Service Annotation Support" in *WebLogic Web Services Reference for Oracle WebLogic Server*.

2	Create a `Dispatch` instance.	See Section 19.6.2, "Creating a Dispatch Instance".
3	Invoke a Web service operation.	You can invoke a Web service operation synchronously (one-way or two-way) or asynchronously (polling or asynchronous handler). See Section 19.6.3, "Invoking a Web Service Operation".

19.6.1 Example of a Web Service Dispatch Client

The following sample shows how to implement a basic Web service Dispatch client. The sample is described in detail in the sections that follow.

Example 19–3 Example of a Web Service Dispatch Client

```
package jaxws.dispatch.client;

import java.io.ByteArrayOutputStream;
import java.io.OutputStream;
import java.io.StringReader;
```

```
import java.net.URL;

import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.WebServiceException;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.namespace.QName;
import javax.xml.ws.soap.SOAPBinding;

public class WebTest extends TestCase {
   private static String in_str = "wiseking";
   private static String request =
      "<ns1:sayHello xmlns:ns1=\"http://example.org\"><arg0>"+in_str+"</arg0></ns1:sayHello>";

   private static final QName portQName = new QName("http://example.org", "SimplePort");
   private Service service = null;

   protected void setUp() throws Exception {

      String url_str = System.getProperty("wsdl");
      URL url = new URL(url_str);
      QName serviceName = new QName("http://example.org", "SimpleImplService");
      service = Service.create(serviceName);
      service.addPort(portQName, SOAPBinding.SOAP11HTTP_BINDING, url_str);
      System.out.println("Setup complete.");

   }

   public void testSayHelloSource() throws Exception {
      setUp();
      Dispatch<Source> sourceDispatch =
         service.createDispatch(portQName, Source.class, Service.Mode.PAYLOAD);
      System.out.println("\nInvoking xml request: " + request);
      Source result = sourceDispatch.invoke(new StreamSource(new StringReader(request)));
      String xmlResult = sourceToXMLString(result);
      System.out.println("Received xml response: " + xmlResult);
      assertTrue(xmlResult.indexOf("HELLO:"+in_str)>=0);
   }

   private String sourceToXMLString(Source result) {
      String xmlResult = null;
      try {
         TransformerFactory factory = TransformerFactory.newInstance();
         Transformer transformer = factory.newTransformer();
         transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
         transformer.setOutputProperty(OutputKeys.METHOD, "xml");
         OutputStream out = new ByteArrayOutputStream();
         StreamResult streamResult = new StreamResult();
         streamResult.setOutputStream(out);
```

```
            transformer.transform(result, streamResult);
            xmlResult = streamResult.getOutputStream().toString();
         } catch (TransformerException e) {
            e.printStackTrace();
         }
         return xmlResult;
      }

}
```

19.6.2 Creating a Dispatch Instance

The `javax.xml.ws.Service` interface acts as a factory for the creation of `Dispatch` instances. So to create a Dispatch instance, you must first create a `Service` instance. Then, create the Dispatch instance using the `Service.createDispatch()` method.

You can pass one or more of the following parameters to the `createDispatch()` method:

- Qualified name (QName) or endpoint reference of the target service endpoint.
- Class of the type parameter `T`. In this example, the `javax.xml.transform.Source` format is used. For valid values, see Table 19–2.
- Usage mode. In this example, the message payload is specified. For valid usage modes, see Table 19–1.
- A list of Web service features to configure on the proxy.
- JAXB context used to marshal or unmarshal messages or message payloads.

For more information about the valid parameters that can be used to call the `Service.createDispatch()` method, see the `javax.xml.ws.Service` Javadoc at:
https://jax-ws.dev.java.net/nonav/2.1.1/docs/api/javax/xml/ws/Service.html.

For example:

```
...
      String url_str = System.getProperty("wsdl");
      QName serviceName = new QName("http://example.org", "SimpleImplService");
      service = Service.create(serviceName);
      service.addPort(portQName, SOAPBinding.SOAP11HTTP_BINDING, url_str);
      Dispatch<Source> sourceDispatch =
        service.createDispatch(portQName, Source.class, Service.Mode.PAYLOAD);
...
```

In the example above, the `createDispatch()` method takes three parameters:

- Qualified name (QName) of the target service endpoint.
- Class of the type parameter `T`. In this example, the `javax.xml.transform.Source` format is used. For valid values, see Table 19–2.
- Usage mode. In this example, the message payload is specified. For valid usage modes, see Table 19–1.

The following example shows how to pass a list of Web service features, specifically the asynchronous client transport feature and asynchronous client handler feature. For

more information about these features, see Chapter 4, "Invoking Web Services Asynchronously."

```
...
  protected Dispatch createDispatch(boolean isSoap12, Class dateType,
                Service.Mode mode, AsyncClientHandlerFeature feature)
                throws Exception {
    String address = publishEndpoint(isSoap12);
    Service service = Service.create(new URL(address + "?wsdl"),
                new QName("http://example.org", "AddNumbersService"));
    QName portName = new QName("http://example.org", "AddNumbersPort");
    AsyncClientTransportFeature transportFeature = new
         AsyncClientTransportFeature("http://localhost:8238/clientsoap12/" +
                UUID.randomUUID().toString());
    Dispatch dispatch = service.createDispatch(portName, dateType, mode,
         feature, transportFeature);
    return dispatch;
  }
...
```

19.6.3 Invoking a Web Service Operation

Once the `Dispatch` instance is created, use it to invoke a Web service operation. You can invoke a Web service operation synchronously (one-way or two-way) or asynchronously (polling or asynchronous handler). For complete details about the synchronous and asynchronous invoke methods, see the `javax.xml.ws.Dispatch` Javadoc at:
https://jax-ws.dev.java.net/nonav/2.1.1/docs/api/javax/xml/ws/Dispatch.html

For example, in the following code excerpt, the XML message is encapsulated as a `javax.xml.transform.stream.StreamSource` object and passed to the synchronous `invoke()` method. The response XML is returned in the *`result`* variable as a `Source` object, and transformed back to XML. The `sourcetoXMLString()` method used to transform the message back to XML is shown in Example 19–3.

```
...
private static String request = "<ns1:sayHello xmlns:ns1=\"http://example.org\"><arg0>"+in_
str+"</arg0></ns1:sayHello>";
Source result = sourceDispatch.invoke(new StreamSource(new StringReader(request)));
String xmlResult = sourceToXMLString(result);
...
```

20

Programming Web Services Using XML Over HTTP

This chapter describes how to program Web services using XML over HTTP.

This chapter includes the following sections:

20.1 About Programming Web Services Using XML Over HTTP

In addition to standard "SOAP over HTTP" use cases, WebLogic JAX-WS can also be used for some "XML over HTTP" Web services. Use of the XML over HTTP style allows you to build simple RESTful Web services while still leveraging the convenience of the JAX-WS programming model.

Note: As a best practice, it is recommended that you develop RESTful Web services using the Jersey JAX-RS RI, as described in *Developing RESTful Web Services*. The Jersey JAX-RS RI provides an open source, production quality RI for building RESTful Web services and supports all of the HTTP methods.

When using the HTTP protocol to access Web service resources, the resource identifier is the URL of the resource and the standard operation to be performed on that resource is one of the HTTP methods: GET, PUT, DELETE, POST, or HEAD.

Note: In this JAX-WS implementation, the set of supported HTTP methods is limited to GET and POST. DELETE, PUT, and HEAD are not supported. Any HTTP requests containing these methods will be rejected with a `405 Method Not Allowed` error.

If the functionality of PUT and DELETE are required, the desired action can be accomplished by tunneling the actual method to be executed on the POST method. This is a workaround referred to as *overloaded POST*. (A Web search on "REST overloaded POST" will return a number of ways to accomplish this.

You build RESTful-like endpoints using the `invoke()` method of the `javax.xml.ws.Provider<T>` interface (see http://download.oracle.com/javaee/5/api/javax/xml/ws/Provider.html). The `Provider` interface provides a dynamic alternative to building an service endpoint interface (SEI).

The procedure in this section describes how to program and compile the JWS file required to implement Web services using XML over HTTP. The procedure shows how to create the JWS file from scratch; if you want to update an existing JWS file, you can also use this procedure as a guide.

It is assumed that you have set up an Ant-based development environment and that you have a working `build.xml` file to which you can add targets for running the `jwsc` Ant task and deploying the Web services. For more information, see *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

Table 20–1 Steps to Program RESTful Web Services

#	Step	Description
1	Create a new JWS file, or update an existing one, that implements the Web service using XML over HTTP.	Use your favorite IDE or text editor. See Section 20.2, "Programming Guidelines for the Web Service Using XML Over HTTP".

2	Update your `build.xml` file to include a call to the `jwsc` Ant task to compile the JWS file into a Web service.	For example:
		`<jwsc srcdir="." destdir="output/restEar">`
		`<jws file="NearbyCity.java" type="JAXWS"/>`
		`</jwsc>`
		For more information, see "Running the jwsc WebLogic Web Services Ant Task" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.

3	Run the Ant target to build the Web service.	For example: `prompt> ant build-rest`
4	Deploy the Web service as usual.	See "Deploying and Undeploying WebLogic Web Services" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
5	Access the Web service from your Web service client.	See Section 20.3, "Accessing the Web Service from a Client".

20.2 Programming Guidelines for the Web Service Using XML Over HTTP

The following example shows a simple JWS file that implements a Web service using XML over HTTP; see the explanation after the example for coding guidelines that correspond to the Java code in **bold**.

```
package examples.webservices.jaxws.rest;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.BindingType;
import javax.xml.ws.Provider;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPBinding;
import javax.xml.ws.http.HTTPException;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.annotation.Resource;
import java.io.ByteArrayInputStream;
```

```
import java.util.StringTokenizer;

@WebServiceProvider(
   targetNamespace="http://example.org",
   serviceName = "NearbyCityService")
@BindingType(value = HTTPBinding.HTTP_BINDING)

public class NearbyCity implements Provider<Source> {
  @Resource(type=Object.class)
  protected WebServiceContext wsContext;

  public Source invoke(Source source)  {
    try {
        MessageContext messageContext = wsContext.getMessageContext();

        // Obtain the HTTP mehtod of the input request.
        javax.servlet.http.HttpServletRequest servletRequest =
          (javax.servlet.http.HttpServletRequest)messageContext.get(
           MessageContext.SERVLET_REQUEST);
        String httpMethod = servletRequest.getMethod();
        if (httpMethod.equalsIgnoreCase("GET"));
        {

            String query =
                 (String)messageContext.get(MessageContext.QUERY_STRING);
            if (query != null && query.contains("lat=") &&
                query.contains("long=")) {
                return createSource(query);
            } else {
                System.err.println("Query String = "+query);
                throw new HTTPException(404);
            }
            } catch(Exception e) {
              e.printStackTrace();
              throw new HTTPException(500);
            }
        }
        } else {
  // This operation only supports "GET"
      throw new HTTPException405);
  }
   private Source createSource(String str) throws Exception {
        StringTokenizer st = new StringTokenizer(str, "=&/");
        String latLong = st.nextToken();
        double latitude = Double.parseDouble(st.nextToken());
        latLong = st.nextToken();
        double longitude = Double.parseDouble(st.nextToken());
        City nearby = City.findNearBy(latitude, longitude);
        String body = nearby.toXML();
        return new StreamSource(new ByteArrayInputStream(body.getBytes()));
   }

   static class City {
        String city;
        String state;
        double latitude;
        double longitude;
        City(String city, double lati, double longi, String st) {
            this.city = city;
            this.state = st;
```

```
            this.latitude = lati;
            this.longitude = longi;
        }

        double distance(double lati, double longi) {
           return Math.sqrt((lati-this.latitude)*(lati-this.latitude) +
               (longi-this.longitude)*(longi-this.longitude)) ;
        }

        static final City[] cities = {
           new City("San Francisco",37.7749295,-122.4194155,"CA"),
           new City("Columbus",39.9611755,-82.9987942,"OH"),
           new City("Indianapolis",39.7683765,-86.1580423,"IN"),
           new City("Jacksonville",30.3321838,-81.655651,"FL"),
           new City("San Jose",37.3393857,-121.8949555,"CA"),
           new City("Detroit",42.331427,-83.0457538,"MI"),
           new City("Dallas",32.7830556,-96.8066667,"TX"),
           new City("San Diego",32.7153292,-117.1572551,"CA"),
           new City("San Antonio",29.4241219,-98.4936282,"TX"),
           new City("Phoenix",33.4483771,-112.0740373,"AZ"),
           new City("Philadelphia",39.952335,-75.163789,"PA"),
           new City("Houston",29.7632836,-95.3632715,"TX"),
           new City("Chicago",41.850033,-87.6500523,"IL"),
           new City("Los Angeles",34.0522342,-118.2436849,"CA"),
           new City("New York",40.7142691,-74.0059729,"NY")};
        static City findNearBy(double lati, double longi) {
            int n = 0;
            for (int i = 1; i < cities.length; i++)  {
                if (cities[i].distance(lati, longi) <
                     cities[n].distance(lati, longi)) {
                    n = i;
                }
            }
            return cities[n];
        }

        public String toXML() {
             return "<ns:NearbyCity xmlns:ns=\"http://example.org\"><City>"
                  +this.city+"</City><State>"+ this.state+"</State><Lat>"
                  +this.latitude +
                  "</Lat><Lng>"+this.longitude+"</Lng></ns:NearbyCity>";
        }
    }
}
```

Follow these guidelines when programming the JWS file that implements the Web service using XML over HTTP. Code snippets of the guidelines are shown in **bold** in the preceding example.

- Import the packages required to implement the Provider Web service.

```
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.BindingType;
import javax.xml.ws.Provider;
```

- Annotate the `Provider` implementation class and set the binding type to HTTP.

```
@WebServiceProvider(
   targetNamespace="http://example.org",
   serviceName = "NearbyCityService")
@BindingType(value = HTTPBinding.HTTP_BINDING)
```

- Implement the `invoke()` method of the `Provider` interface.

```
public class NearbyCity implements Provider<Source> {
  @Resource(type=Object.class)
  protected WebServiceContext wsContext;

  public Source invoke(Source source)  {
  ...
  }
```

- Get the request string using the `QUERY_STRING` field in the `javax.xml.ws.handler.MessageContext` for processing (see message URL http://download.oracle.com/javaee/5/api/javax/xml/ws/handler/MessageContext.html). The query string is then passed to the `createSource()` method that returns the city, state, longitude, and latitude that is closest to the specified values.

```
String query =
             (String)messageContext.get(MessageContext.QUERY_STRING);
.
.
.
return createSource(query);
```

20.3 Accessing the Web Service from a Client

To access the Web service from a Web service client, use the resource URI. For example:

```
URL url = new URL
(http://localhost:7001/NearbyCity/NearbyCityService?lat=35&long=-120);
HttpURLConnection conn = (HttpURLConnection)url.openConnection();
connection.setRequestMethod("POST");
// Get result
InputStream is = connection.getInputStream();
```

In this example, you set the latitude (`lat`) and longitude (`long`) values, as required, to access the required resource.

20.4 Securing Web Services that Use XML Over HTTP

You can secure Web services that use XML over HTTP using the same methods that you use to secure Web applications. For more information, see "Options for Securing Web Application and EJB Resources" in *Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

21

Programming Stateful JAX-WS Web Services Using HTTP Session

This chapter describes how you can develop WebLogic Web services using Java API for XML Web Services (JAX-WS) that interact with an Oracle database.

This chapter includes the following sections:

21.1 Overview of Stateful Web Services

Normally, a JAX-WS Web service is stateless: that is, none of the local variables and object values that you set in the Web service object are saved from one invocation to the next. Even sequential requests from a single client are treated each as independent, stateless method invocations.

There are Web service use cases where a client may want to save data on the service during one invocation and then use that data during a subsequent invocation. For example, a shopping cart object may be added to by repeated calls to the `addToCart` web method and then fetched by the `getCart` web method. In a stateless Web service, the shopping cart object would always be empty, no matter how many `addToCart` methods were called. But by using HTTP Sessions to maintain state across Web service invocations, the cart may be built up incrementally, and then returned to the client.

Enabling stateful support in a JAX-WS Web service requires a minimal amount of coding on both the client and server.

21.2 Accessing HTTP Session on the Server

On the server, every Web service invocation is tied to an HttpSession object. This object may be accessed from the Web service Context that, in turn, may be bound to the Web service object using resource injection. Once you have access to your HttpSession object, you can "hang" off of it any stateful objects you want. The next time your client calls the Web service, it will find that same HttpSession object and be able to lookup the objects previously stored there. Your Web service is stateful!

The steps required on the server:

1. Add the @Resource (defined by Common Annotations for the Java Platform, JSR 250) to the top of your Web service.
2. Add a variable of type WebServiceContext that will have the context injected into it.
3. Using the Web service context, get the HttpSession object.
4. Save objects in the HttpSession using the setAttribute method and retrieve saved object using getAttribute. Objects are identified by a string value you assign.

The following snippet shows its usage:

Example 21–1 Accessing HTTP Session on the Server

```
@WebService
public class ShoppingCart {
   @Resource    // Step 1
   private WebServiceContext wsContext;    // Step 2
   public int addToCart(Item item) {
      // Find the HttpSession
      MessageContext mc = wsContext.getMessageContext();    // Step 3
      HttpSession session =
((javax.servlet.http.HttpServletRequest)mc.get(MessageContext.SERVLET_
REQUEST)).getSession();
      if (session == null)
         throw new WebServiceException("No HTTP Session found");
      // Get the cart object from the HttpSession (or create a new one)
      List<Item> cart = (List<Item>)session.getAttribute("myCart");  // Step 4
      if (cart == null)
         cart = new ArrayList<Item>();
      // Add the item to the cart (note that Item is a class defined
      // in the WSDL)
      cart.add(item);
      // Save the updated cart in the HTTPSession (since we use the same
      // "myCart" name, the old cart object will be replaced)
      session.setAttribute("myCart", cart);
      // return the number of items in the stateful cart
      return cart.size();
   }
}
```

21.3 Enabling HTTP Session on the Client

The client-side code is quite simple. All you need to do is set the SESSION_ MAINTAIN_PROPERTY on the request context. This tells the client to pass back the HTTP Cookies that it receives from the Web service. The cookie contains a session ID that allows the server to match the Web service invocation with the correct HttpSession, providing access to any saved stateful objects.

Example 21–2 Enabling HTTP Session on the Client

```
ShoppingCart proxy = new CartService().getCartPort();
((BindingProvider)proxy).getRequestContext().put(BindingProvider.SESSION_MAINTAIN_
PROPERTY, true);
// Create a new Item object with a part number of '123456' and an item
// count of 4.
Item item = new Item('123456', 4);
// After first call, we'll print '1' (the return value is the number of objects
// in the Cart object)
System.out.println(proxy.addToCart(item));
```

```
// After the second call, we'll print '2', since we've added another
// Item to the stateful, saved Cart object.
System.out.println(proxy.addToCart(item));
```

21.4 Developing Stateful Services in a Cluster Using Session State Replication

In a high-availability environment, a JAX-WS Web service may be replicated across multiple server instances in a cluster. A stateful JAX-WS Web service is supported in this environment through the use of the WebLogic Server HTTP Session State Replication feature. For more information, see "HTTP Session State Replication" in *Using Clusters for Oracle WebLogic Server*.

There are a variety of techniques and configuration requirements for setting up a clustered environment using session state replication (for example, supported servers and load balancers, and so on). From the JAX-WS programming perspective, the only new consideration is that the objects you store in the HttpSession using the HttpSession.setAttribute method (as in Example 21–1) must be Serializable. If they are Serializable, then these stateful objects become available to the Web service on all replicated Web service instances in the cluster, providing both load balancing and failover capabilities for JAX-WS stateful Web services.

21.5 A Note About the JAX-WS RI @Stateful Extension

The JAX-WS 2.1 Reference Implementation (RI) contains a vendor extension that supports a different model for stateful JAX-WS Web services using the @Stateful annotation. It's implementation "pins" the state to a particular instance and is not designed to be scalable or fault-tolerant. This feature is not supported for WebLogic Server JAX-WS Web services.

A

Pre-packaged WS-Policy Files for Web Services Reliable Messaging and Make Connection

This appendix summarizes the pre-packaged WS-Policy files that support reliable messaging, Make Connection, or both features together, for WebLogic Web services using Java API for XML Web Services (JAX-WS).

You cannot change these pre-packaged files. If their values do not suit your needs, you must create your own WS-Policy file. For details, see:

- Section 6.4, "Creating the Web Service Reliable Messaging WS-Policy File"
- Section 4.6.1.1, "Creating the Web Service Make Connection WS-Policy File (Optional)"

For reference information about the reliable messaging and Make Connection policy assertions, see:

- "Web Service Reliable Messaging Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*
- "Web Service Make Connection Policy Assertion Reference" in *WebLogic Web Services Reference for Oracle WebLogic Server*

The following table summarizes the pre-packaged WS-Policy files. This table also specifies whether the WS-Policy file can be attached at the method level; if the value in this column is no, then the WS-Policy file can be attached at the class level only.

Table A–1 Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`DefaultReliability1.2.xml`	Specifies policy assertions related to delivery assurance. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Section A.1, "DefaultReliability1.2.xml (WS-Policy File)".	Yes
`DefaultReliability1.1.xml`	Specifies policy assertions related to quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Section A.2, "DefaultReliability1.1.xml (WS-Policy File)".	Yes
`DefaultRelibility.xml`	**Deprecated.** The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at `http://schemas.xmlsoap.org/ws/2005/02/rm/WS-RMPolicy.pdf`. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. Specifies typical values for the reliable messaging policy assertions, such as inactivity timeout of 10 minutes, acknowledgement interval of 200 milliseconds, and base retransmission interval of 3 seconds. See Section A.3, "DefaultReliability.xml WS-Policy File (WS-Policy) [Deprecated]".	Yes
`LongRunningReliability.xml`	**Deprecated.** The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 for long running processes. In this release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration. Similar to the preceding default reliable messaging WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours.) See Section A.4, "LongRunningReliability.xml WS-Policy File (WS-Policy) [Deprecated]".	Yes
`Mc1.1.xml`	Enables Make Connection support on the Web service and specifies usage as optional on the Web service client. The WS-Policy 1.5 protocol is used. See Section A.5, "Mc1.1.xml (WS-Policy File)".	No
`Mc.xml`	Enables Make Connection support on the Web service and specifies usage as optional on the Web service client. The WS-Policy 1.2 protocol is used. See Section A.6, "Mc.xml (WS-Policy File)".	No
`Reliability1.2_ExactlyOnce_WithMC1.1.xml`	Specifies policy assertions related to quality of service. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client. See Section A.7, "Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File)".	No

Table A–1 (Cont.) Pre-packaged WS-Policy Files That Support Reliable Messaging

Pre-packaged WS-Policy File	Description	Method Level Attachment?
`Reliability1.2_SequenceSTRSecurity`	Specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Section A.10, "Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File)".	No
`Reliability1.1_SequenceSTRSecurity`	The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Section A.11, "Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)"	Yes
`Reliability1.2_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702`. See Section A.10, "Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File)".	Yes
`Reliability1.1_SequenceTransportSecurity`	Specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at `http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html`. See Section A.11, "Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)".	Yes
`Reliability1.0_1.2.xml`	Combines 1.2 and 1.0 WS-Reliable Messaging policy assertions. The policy assertions for the 1.2 version Make Connection support on the Web service and specifies usage as optional on the Web service client. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. See Section A.12, "Reliability1.0_1.2.xml (WS-Policy File)".	No
`Reliability1.0_1.1.xml`	Combines 1.1 and 1.0 WS Reliable Messaging policy assertions. See Section A.13, "Reliability1.0_1.1.xml (WS-Policy.xml File)".	Yes

A.1 DefaultReliability1.2.xml (WS-Policy File)

The `DefaultRealiability1.2.xml` WS-Policy file specifies policy assertions related to delivery assurance. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
```

```
            <wsrmp:DeliveryAssurance>
               <wsp15:Policy>
                  <wsrmp:ExactlyOnce/>
                  <wsrmp:InOrder/>
               </wsp15:Policy>
            </wsrmp:DeliveryAssurance>
         </wsrmp:RMAssertion>
      </wsp15:All>
   </wsp15:Policy>
```

A.2 DefaultReliability1.1.xml (WS-Policy File)

The `DefaultRealiability1.1.xml` WS-Policy file specifies policy assertions related to quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html.

```
<?xml version="1.0"?>

<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  >
  <wsrmp:RMAssertion
    xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702"
    >
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce />
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

A.3 DefaultReliability.xml WS-Policy File (WS-Policy) [Deprecated]

This WS-Policy file is deprecated. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at http://schemas.xmlsoap.org/ws/2005/02/rm/policy/. In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The `DefaultReliability.xml` WS-Policy file specifies typical values for the reliable messaging policy assertions, such as inactivity timeout of 10 minutes, acknowledgement interval of 200 milliseconds, and base retransmission interval of 3 seconds.

```
<?xml version="1.0"?>

<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy"
  >

  <wsrm:RMAssertion >
    <wsrm:InactivityTimeout Milliseconds="600000" />
```

```
    <wsrm:BaseRetransmissionInterval Milliseconds="3000" />
    <wsrm:ExponentialBackoff />
    <wsrm:AcknowledgementInterval Milliseconds="200" />
    <beapolicy:Expires Expires="P1D" optional="true"/>
  </wsrm:RMAssertion>
</wsp:Policy>
```

A.4 LongRunningReliability.xml WS-Policy File (WS-Policy) [Deprecated]

This WS-Policy file is deprecated. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion Version 1.0 at http://schemas.xmlsoap.org/ws/2005/02/rm/policy/. In the current release, many of the reliable messaging policy assertions are managed through JWS annotations or configuration.

The `LongRunningRelibility.xml` WS-Policy files specifies values that are similar to the `DefaultReliability.xml` WS-Policy file, except that it specifies a much longer activity timeout interval (24 hours). See Section A.4, "LongRunningReliability.xml WS-Policy File (WS-Policy) [Deprecated]".

```
<?xml version="1.0"?>

<wsp:Policy
   xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:beapolicy="http://www.bea.com/wsrm/policy"
  >
  <wsrm:RMAssertion >
    <wsrm:InactivityTimeout Milliseconds="86400000" />
    <wsrm:BaseRetransmissionInterval Milliseconds="3000" />
    <wsrm:ExponentialBackoff />
    <wsrm:AcknowledgementInterval Milliseconds="200" />
    <beapolicy:Expires Expires="P1M" optional="true"/>
  </wsrm:RMAssertion>
</wsp:Policy>
```

A.5 Mc1.1.xml (WS-Policy File)

The `Mc1.1.xml` WS-Policy file enables Make Connection support on the Web service and sets usage as optional on the Web service client. In this case, the WS-Policy 1.5 protocol is used. The assertions are based on the Make Connection policy assertion defined at http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html.

```
<?xml version="1.0"?>
<wsp15:Policy
  xmlns:wsp15="http://www.w3.org/ns/ws-policy"
  xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702">
  <wsmc:MCSupported wsp15:Optional="true" />
</wsp15:Policy>
```

A.6 Mc.xml (WS-Policy File)

The `Mc.xml` WS-Policy file enables Make Connection support on the Web service and sets usage as optional on the Web service client. The assertions are based on the Make Connection policy assertion defined at http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html.

```
<?xml version="1.0"?>
<wsp:Policy
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702">
  <wsmc:MCSupported wsp:Optional="true" />
</wsp:Policy>
```

A.7 Reliability1.2_ExactlyOnce_WithMC1.1.xml (WS-Policy File)

The `Reliability1.2_ExactlyOnce_WithMC1.1.xml` WS-Policy file specifies policy assertions related to quality of service. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client.

The assertions are based on the following specifications:

- Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html.
- Make Connection assertions are based on the Make Connection policy assertion defined at http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html.

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
       xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
         <wsrmp:DeliveryAssurance>
            <wsp15:Policy>
               <wsrmp:ExactlyOnce />
            </wsp15:Policy>
         </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
      <wsmc:MCSupported
        xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702"
        wsp15:Optional="true" />
   </wsp15:All>
</wsp15:Policy>
```

A.8 Reliability1.2_SequenceSTR.xml (WS-Policy File)

The `Reliability1.2_SequenceSTR.xml` file specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. It enables Make Connection support on the Web service and specifies usage as optional on the Web service client.

The assertions are based on the following specifications:

- Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2 at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html.
- Make Connection assertions are based on the Make Connection policy assertion defined at http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.1-spec-os.html.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
       xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
         <wsrmp:SequenceSTR/>
         <wsrmp:DeliveryAssurance>
            <wsp15:Policy>
               <wsrmp:ExactlyOnce/>
            </wsp15:Policy>
         </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
      <wsmc:MCSupported
       xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702"
       wsp15:Optional="true"/>
   </wsp15:All>
</wsp15:Policy>
```

A.9 Reliability1.1_SequenceSTR.xml (WS-Policy File)

The `Reliability1.1_SequenceSTR.xml` file specifies that in order to secure messages in a reliable sequence, the runtime will use the `wsse:SecurityTokenReference` that is referenced in the `CreateSequence` message. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsrmp:RMAssertion
         xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceSTR/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

A.10 Reliability1.2_SequenceTransportSecurity.xml (WS-Policy File)

The `Reliability1.2_SequenceTransportSecurity.xml` file specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.2

at
http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:All>
      <wsrmp:RMAssertion
       xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
         <wsrmp:SequenceTransportSecurity/>
            <wsrmp:DeliveryAssurance>
               <wsp15:Policy>
                  <wsrmp:ExactlyOnce/>
               </wsp15:Policy>
            </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
   </wsp15:All>
</wsp15:Policy>
```

A.11 Reliability1.1_SequenceTransportSecurity.xml (WS-Policy File)

The `Reliability1.1_SequenceTransportSecurity.xml` file specifies policy assertions related to transport-level security and quality of service. The Web service reliable messaging assertions are based on WS Reliable Messaging Policy Assertion 1.1 at
http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
    <wsrmp:RMAssertion
        xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
    <wsrmp:SequenceTransportSecurity/>
    <wsrmp:DeliveryAssurance>
      <wsp:Policy>
        <wsrmp:ExactlyOnce/>
      </wsp:Policy>
    </wsrmp:DeliveryAssurance>
  </wsrmp:RMAssertion>
</wsp:Policy>
```

A.12 Reliability1.0_1.2.xml (WS-Policy File)

The `Reliability1.0_1.2.xml` WS-Policy file combines 1.2 and 1.0 WS-Reliable Messaging policy assertions.

This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. For more information about smart policy selection, see Section 6.4.3, "Using Multiple Policy Alternatives".

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp15:Policy xmlns:wsp15="http://www.w3.org/ns/ws-policy">
   <wsp15:ExactlyOne>
      <wsp15:All>
         <wsrmp:RMAssertion
          xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
            <wsrmp:DeliveryAssurance>
               <wsp15:Policy>
                  <wsrmp:ExactlyOnce/>
```

```
            </wsp15:Policy>
          </wsrmp:DeliveryAssurance>
        </wsrmp:RMAssertion>
        <wsmc:MCSupported
          xmlns:wsmc="http://docs.oasis-open.org/ws-rx/wsmc/200702"
          wsp15:Optional="true"/>
      </wsp15:All>
      <wsp15:All>
        <wsrmp10:RMAssertion
         xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
          <wsrmp10:InactivityTimeout Milliseconds="600000"/>
          <wsrmp10:BaseRetransmissionInterval Milliseconds="3000"/>
          <wsrmp10:ExponentialBackoff/>
          <wsrmp10:AcknowledgementInterval Milliseconds="200"/>
        </wsrmp10:RMAssertion>
      </wsp15:All>
   </wsp15:ExactlyOne>
</wsp15:Policy>
```

A.13 Reliability1.0_1.1.xml (WS-Policy.xml File)

The `Reliability1.0_1.1.xml` WS-Policy file combines 1.1 and 1.0 WS-Reliable Messaging policy assertions. This sample relies on smart policy selection to determine the policy assertion that is applied at runtime. For more information about smart policy selection, see Section 6.4.3, "Using Multiple Policy Alternatives".

> **Note:** The 1.0 Web service reliable messaging assertions are prefixed by `wsrmp10`.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsrmp:RMAssertion
           xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">
        <wsrmp:DeliveryAssurance>
          <wsp:Policy>
            <wsrmp:ExactlyOnce/>
          </wsp:Policy>
        </wsrmp:DeliveryAssurance>
      </wsrmp:RMAssertion>
    </wsp:All>
    <wsp:All>
      <wsrmp10:RMAssertion
       xmlns:wsrmp10="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp10:InactivityTimeout Milliseconds="600000"/>
        <wsrmp10:BaseRetransmissionInterval Milliseconds="3000"/>
        <wsrmp10:ExponentialBackoff/>
        <wsrmp10:AcknowledgementInterval Milliseconds="200"/>
      </wsrmp10:RMAssertion>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

B

Example Client Wrapper Class for Batching Reliable Messages

This appendix provides an example client wrapper class that can be used for batching reliable messaging for WebLogic Web services using Java API for XML Web Services (JAX-WS).

For more information about batching reliable messages, see Section 6.11, "Grouping Messages into Business Units of Work (Batching)."

> **Note:** This client wrapper class is example code only; it is not an officially supported production class.

Example B–1 Example Client Wrapper Class for Batching Reliable Messages

```
package example.servlet;

import java.io.PrintStream;
import java.io.PrintWriter;
import java.lang.ref.WeakReference;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Date;
import java.util.SortedSet;
import java.util.Timer;
import java.util.TimerTask;

import javax.xml.datatype.DatatypeFactory;
import javax.xml.datatype.Duration;
import javax.xml.ws.BindingProvider;

import weblogic.wsee.jaxws.JAXWSProperties;
import weblogic.wsee.jaxws.spi.ClientInstance;
import weblogic.wsee.reliability.MessageRange;
import weblogic.wsee.reliability2.api.WsrmClient;
import weblogic.wsee.reliability2.api.WsrmClientFactory;
import weblogic.wsee.reliability2.property.WsrmInvocationPropertyBag;
import weblogic.wsee.reliability2.tube.WsrmClientImpl;

/**
 * Example wrapper class to batch reliable requests into fixed size 'batches'
 * that can be sent using a single RM sequence. This class allows a client to
 * send requests that have no natural common grouping or
 * 'business unit of work' and not pay the costs associated with creating and
```

```
 * terminating an RM sequence for every message.
 * NOTE: This class is an *example* of how batching might be performed. To do
 *       batching correctly, you should consider error recovery and how to
 *       report sequence errors (reported via ReliabilityErrorListener) back
 *       to the clients that made the original requests.
 * <p>
 * If your Web service client code knows of some natural business-oriented
 * grouping of requests (called a 'business unit of work'), it should make the
 * RM subsystem aware of this unit of work by using the
 * WsrmClient.setFinalMessage() method to demarcate the end of a unit (just
 * before sending the actual final request via an invocation on
 * the client instance). In some cases, notably when the client code represents
 * an intermediary in the processing of messages, the client code may not be
 * aware of any natural unit of work. In the past, if no business unit of work
 * could be determined, clients often just created the client instance, sent the
 * single current message they had, and then allowed the sequence to terminate.
 * This is functionally workable, but very inefficient. These clients pay the
 * cost of an RM sequence handshake and termination for every message they send.
 * The BatchingRmClientWrapper class can be used to introduce an artificial
 * unit of work (a batch) when no natural business unit of work is available.
 * <p>
 * Each instance of BatchingRmClientWrapper is a wrapper instance around a
 * client instance (port or Dispatch instance). This wrapper can be used to
 * obtain a Proxy instance that can be used in place of the original client
 * instance. This allows this class to perform batching operations completely
 * invisibly from the perspective of the client code.
 * <p>
 * This class is used for batching reliable requests into
 * batches of a given max size that will survive for a given maximum
 * duration. If a batch fills up or times out, it is ended, causing the
 * RM sequence it represents to be ended/terminated. The timeout ensures that
 * if the flow of incoming requests stops the batch/sequence will still
 * end in a timely manner.
 */
public class BatchingRmClientWrapper<T>
  implements InvocationHandler {

  private Class<T> _clazz;
  private int _batchSize;
  private long _maxBatchLifetimeMillis;
  private T _clientInstance;
  private PrintWriter _out;
  private WsrmClient _rmClient;
  private int _numInCurrentBatch;
  private int _batchNum;
  private Timer _timer;
  private boolean _closed;
  private boolean _proxyCreated;

  /**
   * Create a wrapper instance for batching reliable requests into
   * batches of the given max size that will survive for the given maximum
   * duration. If a batch fills up or times out, it is ended, causing the
   * RM sequence it represents to be ended/terminated.
   * @param clientInstance The client instance that acts as the source object
   *        for the batching proxy created by the createProxy() method. This
   *        is the port/Dispatch instance returned from the call to
   *        getPort/createDispatch. The BatchingRmClientWrapper will take over
   *        responsibility for managing the interaction with and cleanup of
   *        the client instance via the proxy created from createProxy.
```

```
 * @param clazz of the proxy instance we'll be creating in createProxy.
 *        This should be the class of the port/Dispatch instance you would
 *        use to invoke operations on the service. BatchingRmClientWrapper will
 *        create (via createProxy) a proxy of the given type that can be
 *        used in place of the original client instance.
 * @param batchSize Max number of requests to put into a batch. If the
 *        max number of requests are sent for a given batch, that batch is
 *        ended (ending/terminating the sequence it represents) and a new
 *        batch is started.
 * @param maxBatchLifetime A duration value (in the lexical form supported
 *        by java.util.Duration, e.g. PT30S for 30 seconds) representing
 *        the maximum time a batch should exist. If the batch exists longer
 *        than this time, it is ended and a new batch is begun.
 * @param out A print stream that can be used to print diagnostic and
 *        status messages.
 */
public BatchingRmClientWrapper(T clientInstance, Class<T> clazz,
                               int batchSize, String maxBatchLifetime,
                               PrintStream out) {
  _clazz = clazz;
  _batchSize = batchSize;
  try {
    if (maxBatchLifetime == null) {
      maxBatchLifetime = "PT5M";
    }
    Duration duration =
      DatatypeFactory.newInstance().newDuration(maxBatchLifetime);
    _maxBatchLifetimeMillis = duration.getTimeInMillis(new Date());
  } catch (Exception e) {
    throw new RuntimeException(e.toString(), e);
  }
  _clientInstance = clientInstance;
  _out = new PrintWriter(out, true);
  _rmClient = WsrmClientFactory.getWsrmClientFromPort(_clientInstance);
  _closed = false;
  _proxyCreated = false;
  _timer = new Timer(true);
  _timer.schedule(new TimerTask() {
    @Override
    public void run() {
      terminateOrEndBatch();
    }
  }, _maxBatchLifetimeMillis);
}

/**
 * Creates the dynamic proxy that should be used in place of the client
 * instance used to create this BatchingRmClientWrapper instance. This method
 * should be called only once per BatchingRmClientWrapper.
 */
public T createProxy() {
  if (_proxyCreated) {
    throw new IllegalStateException("Already created the proxy for this BatchingRmClientWrapper
      instance which wraps the client instance: " + _clientInstance);
  }
  _proxyCreated = true;
  return (T) Proxy.newProxyInstance(getClass().getClassLoader(),
                                    new Class[] {
                                      _clazz,
                                      BindingProvider.class,
```

```
                                    java.io.Closeable.class
                                  }, this);
  }

  private void terminateOrEndBatch() {
    synchronized(_clientInstance) {
      if (_rmClient.getSequenceId() != null) {
        if (terminateBatchAllowed()) {
          _out.println("Terminating batch " + _batchNum + " sequence (" + _
                rmClient.getSequenceId() + ") for " + _clientInstance);
          try {
            _rmClient.terminateSequence();
          } catch (Exception e) {
            e.printStackTrace(_out);
          }
        } else {
          _out.println("Batch " + _batchNum + " sequence (" + _rmClient.getSequenceId() + ")
            for " + _clientInstance + " timed out but has outstanding requests to send and
            cannot be terminated now");
        }
      }
      endBatch();
    }
  }

  /**
   * Check to see if we have acks for all requests sent. If so,
   * we can terminate.
   */
  private boolean terminateBatchAllowed() {
    try {
      synchronized(_clientInstance) {
        if (_rmClient.getSequenceId() != null) {
          long maxMsgNum = _rmClient.getMostRecentMessageNumber();
          if (maxMsgNum < 1) {
            // No messages sent, go ahead and terminate.
            return true;
          }
          SortedSet<MessageRange> ranges = _rmClient.getAckRanges();
          long maxAck = -1;
          boolean hasGaps = false;
          long lastRangeUpper = -1;
          for (MessageRange range: ranges) {
            if (lastRangeUpper > 0) {
              if (range.lowerBounds != lastRangeUpper + 1) {
                hasGaps = true;
              }
            } else {
              lastRangeUpper = range.upperBounds;
            }
            maxAck = range.upperBounds;
          }
          return !(hasGaps || maxAck < maxMsgNum);
        }
      }
    } catch (Exception e) {
      e.printStackTrace(_out);
    }
    return true;
  }
```

```
private void endBatch() {
  synchronized(_clientInstance) {
    if (_numInCurrentBatch > 0) {
      _out.println("Ending batch " + _batchNum + " sequence (" + _rmClient.getSequenceId() + ")
         for " + _clientInstance + "...");
    }
    /**
     * _rmClient.reset() resets a WsrmClient instance (and the client instance it represents)
     * so it can track a new WS-RM sequence for the next invoke on the client
     * instance. This method effectively *disconnects* the RM sequence from the
     * client instance and lets them continue/complete separately.
     */
    _rmClient.reset();
    _numInCurrentBatch = 0;
    if (!_closed) {
      _timer.schedule(new TimerTask() {
        @Override
        public void run() {
          terminateOrEndBatch();
        }
      }, _maxBatchLifetimeMillis);
    }
  }
}

public Object invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
  boolean operationInvoke = method.getDeclaringClass() == _clazz;
  boolean closeableInvoke = method.getDeclaringClass() ==
                            java.io.Closeable.class;
  boolean endOfBatch = false;
  if (operationInvoke) {
    synchronized(_clientInstance) {
      // Check our batch size
      if (_numInCurrentBatch == 0) {
        _batchNum++;
      }
      endOfBatch = _numInCurrentBatch >= _batchSize - 1;
      if (endOfBatch) {
        _rmClient.setFinalMessage();
      }
      _out.println("Making " + (endOfBatch ? "final " : "") + "invoke " +
         (_numInCurrentBatch+1) + " of batch " + _batchNum + " sequence (" + _
         rmClient.getSequenceId() + ") with operation: " + method.getName());
    }
  } else if (closeableInvoke && method.getName().equals("close")) {
    synchronized(_clientInstance) {
      // Make sure we don't try to schedule the timer anymore
      _closed = true;
      _timer.cancel();
    }
  }
  Object ret = method.invoke(_clientInstance, args);
  if (operationInvoke) {
    synchronized(_clientInstance) {
      _numInCurrentBatch++;
      if (endOfBatch) {
        endBatch();
      }
```

```
      }
    }
    return ret;
  }
}
```